O'REILLY

TURING 图灵程序设计丛书

Docker 经典实例

Docker Cookbook

超过130个经过验证的Docker实践

直指容器精髓



[美] Sébastien Goasguen 著

刘斌 译

中国工信出版集团

人民邮电出版社
POSTS & TELECOM PRESS

数字版权声明

图灵社区的电子书没有采用专有客户端，您可以在任意设备上，用自己喜欢的浏览器和PDF阅读器进行阅读。

但您购买的电子书仅供您个人使用，未经授权，不得进行传播。

我们愿意相信读者具有这样的良知和觉悟，与我们共同保护知识产权。

如果购买者有侵权行为，我们可能对该用户实施包括但不限于关闭该帐号等维权措施，并可能追究法律责任。

译者介绍

刘斌

具有10余年软件开发经验，关注后台开发技术和各种编程语言。做过电子商务、金融、企业系统和Android手机开发，写过Delphi，也兼做系统管理员和DBA，最近在做与Docker相关的工作。

个人主页：http://liubin.org

微信公众号：西小口物语（xxk_talks）

TURING 图灵程序设计丛书

Docker经典实例

Docker Cookbook

[美] Sébastien Goasguen 著
刘斌 译

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

O'Reilly Media, Inc.授权人民邮电出版社出版

人民邮电出版社
北 京

图书在版编目（C I P）数据

Docker经典实例 / (美) 戈阿冈著 ; 刘斌译. -- 北
京 : 人民邮电出版社, 2017.2
（图灵程序设计丛书）
ISBN 978-7-115-44656-5

Ⅰ. ①D… Ⅱ. ①戈… ②刘… Ⅲ. ①Linux操作系统
—程序设计 Ⅳ. ①TP316.85

中国版本图书馆CIP数据核字(2017)第005760号

内 容 提 要

本书结构明晰，示例丰富详实，是全面实用的 Docker 入门教程。作者全面介绍了 Docker 相关各种工具和平台，涵盖网络、镜像管理、配置以及包括 Kubernetes 和 Mesos 在内的编排和调度生态系统，对私有云和公有云上部署的应用程序都给出了丰富实用的解决方案和示例。

本书适合运维人员、系统管理员和开发人员阅读。

◆ 著　　[美] Sébastien Goasguen
译　　刘 斌
责任编辑　朱 巍
执行编辑　贺子娟 赵瑞琳
责任印制　彭志环

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
邮编 100164　　电子邮件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
北京　　　　印刷

◆ 开本：800×1000 1/16
印张：19.75
字数：468千字　　2017年 2 月第 1 版
印数：1 – 3 000册　　2017年 2 月北京第 1 次印刷

著作权合同登记号 图字：01-2016-6549号

定价：69.00元

读者服务热线：(010)51095186转600 印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京东工商广字第 8052 号

版权声明

O’Reilly Media, Inc.介绍

O’Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O’Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O’Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O’Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 *Make* 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O’Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O’Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过图书出版、在线服务或者面授课程，每一项 O’Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O’Reilly Radar 博客有口皆碑。”

——*Wired*

“O’Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——*Business 2.0*

“O’Reilly Conference 是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本 O’Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野，并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去，Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——*Linux Journal*

目录

本书赞誉

开始使用 Docker 是一回事，真正领悟它的思想又是另一回事。我们需要对 Docker 有一个完整深入的理解。在为用户提供服务的应用程序中使用 Docker 时，这本书为我们带来了极大的帮助。

——Arjan Eriks，Schuberg Philis 公司云计算服务主管

这是一部完整实用的教程，涵盖了与 Docker 相关的各种工具和平台，并提供了具体和实用的例子。由于 Docker 的核心功能通过开放容器计划（Open Container Initiative）的努力逐渐成为事实上的行业标准，我们可以预想到，这个生态系统还会继续快速扩张。Sébastien 的这本书为从业者打下了坚实的基础，使得他们可以跟上这种快速变化的步伐。

——Chip Childers，Cloud Foundry 基金会技术副总裁

Sébastien 做了一项非常棒的工作，为初级用户集中介绍了各种 Docker 最佳实践和入门材料，涵盖了网络、镜像管理、配置以及包括 Kubernetes 和 Mesos/Marathon 在内的正在快速发展中的编排和调度生态系统。

——Patrick Reilly，Kismatic 公司 CEO

前言

写作缘由

我已经在云计算领域（主要是 IaaS 层）工作了 10 余年。Amazon AWS、Google GCE 和 Microsoft Azure 提供大规模的云计算服务已经有几年了，毫不夸张地说，访问一台服务器从未像现在这样方便、快速。对我来说，其真正的价值在于可以通过 API 来访问这些服务。我们现在可以通过编程来创建基础设施和部署应用。这些可编程层能够帮助我们达到更高级别的自动化，有利于企业更快地将产品推向市场，做出更多的创新，以及更好地为用户服务。

然而，尽管我们在配置管理和编排上耗费了大量的精力，但是对于在一个分布式环境中进行应用程序打包、配置和服务组装等方面依旧没有取得太大的进步。部署和运行一个可扩展、可容错的分布式应用程序仍然是比较困难的。

直到我试用 Docker 并明白了它为我们带来的可能性之后，我才成为 Docker 的疯狂粉丝。Docker 为 Linux 容器带来了全新的用户体验。它不是要提供与容器相对的完全虚拟化技术，而是要为应用程序的打包和运行提供便利。一旦你开始使用 Docker 并享受它所带来的全新体验，就会同时体会到另一个好处：你会开始思考如何进行合成和聚类分析。

容器帮助我们更多地思考如何进行*功能隔离*，这反过来又迫使我们在分布式环境中对应用进行解耦，然后再将其粘合在一起。

本书结构

本书共由 10 章构成。每章都基于 O'Reilly 标准的 cookbook 格式写成，由问题、解决方案和讨论等三部分组成。你可以按照顺序从前往后阅读，也可以选择特定的章节 / 范例。每个范例都是互相独立的，但如果一个范例引用了其他范例的概念，书中也会提供相应的参考引用。

- 第 1 章主要讨论与 Docker 安装相关的一些场景，包括使用 Docker Machine。然后会介绍 Docker 的一些基本命令，包括对容器进行管理、挂载数据卷、链接容器等。在该章最后，你将会得到一个可以工作的 Docker 主机，启动一些容器，还会对容器的生命周期有所了解。
- 第 2 章将会介绍 Dockerfile 和 Docker Hub，并展示如何构建镜像，以及对镜像进行打标签和提交操作。该章还将介绍如何运行自己的 Docker registry，并设置自动构建。学完该章，你将会掌握如何创建 Docker 镜像，通过公开或者私有的方式来共享镜像，以及构建持续交付工作流。
- 第 3 章将介绍 Docker 的网络机制。你将学到如何获得一个容器的 IP 地址，以及如何暴露主机端口上的容器服务。你还将学会如何把多个容器链接起来，以及如何使用定制的网络配置。该章也会通过一些范例来对容器网络进行深入剖析。对网络命名空间、使用 OVS 网桥、GRE 隧道等概念的介绍，也将为了解 Docker 容器网络奠定基础。最后，你还将了解更高级的网络配置和工具，比如 Weave、Flannel 以及目前处于实验阶段的 Docker Network 功能。
- 第 4 章将深入介绍如何对 Docker 守护进程进行配置，特别是与 Docker API 相关的安全设置和远程访问控制。该章也讨论了一些基本问题，如从源代码编译 Docker，运行 Docker 的测试用例，以及运行自己编译的 Docker 可执行程序。该章还有一些范例，对 Linux 上的命名空间和它们在容器中的使用情况进行了详细阐述。
- 第 5 章将会介绍 Google 新开发的容器管理平台。Kubernetes 提供了一种在分布式集群上部署多容器应用程序的方式。此外，它还提供了一种自动化公开服务和创建容器副本的方式。该章将会介绍如何在自己的基础设施上部署 Kubernetes：开始是使用本地 Vagrant 集群，随后是在云端启动的一组计算机上。之后，我们将介绍 Kubernetes 的核心内容：pod、service 和 replication controller。
- 第 6 章涵盖了四种新的经过优化的、专门为运行容器而生的 Linux 发行版本：CoreOS（https://coreos.com/）、Project Atomic（http://www.projectatomic.io/）、Ubuntu Core（http://www.ubuntu.com/cloud/tools/snappy） 和 RancherOS（http://rancher.com/rancher-os/）。这些新的发行版本刚好为 Docker 容器的运行和编排提供了够用的操作系统。该章的范例包括如何安装和访问采用了这些发行版本的服务器。该章还会介绍这些发行版本所使用的对容器进行编排的工具，例如 `etcd`、`fleet` 和 `systemd`。
- 蓬勃发展的生态系统也是 Docker 的优势之一。第 7 章将会介绍过去 18 个月中出现的一些新工具。这些工具用于帮助 Docker 进行应用程序部署、持续集成、服务发现和编排。举个例子，你会读到关于 Docker Compose、Docker Swarm、Mesos、Rancher 和 Weavescope 的范例。
- Docker 守护进程可以安装在本地开发计算机上。然而，随着云计算技术的出现，我们可以按需创建服务器并立时使用。毫不夸张地说，大量基于容器的应用程序将会被部署到云中。第 8 章中的范例将会介绍如何访问位于 Amazon AWS（http://aws.amazon.com/）、Google GCE（https://cloud.google.com/compute/）和 Microsoft Azure（http://azure.microsoft.com/en-us/）之上的 Docker 主机。该章还将介绍两种使用了 Docker 的新云计算服务：AWS 弹性容器服务（Elastic Container Service，ECS）和 Google 容器引擎（Google Container Engine，https://cloud.google.com/container-engine/）。

- 第 9 章将会探讨容器使用过程中的应用程序监控问题。长久以来，基础设施和应用程序的监控和可视化一直是 DevOps 社区关注的重点。随着 Docker 作为一种开发和运维机制变得越来越普及，从其中所汲取的经验教训也需要应用于基于容器的应用程序。
- 第 10 章将介绍单主机和集群上的端到端应用的部署。虽然前面几章中已经介绍了一些基本的应用程序部署，但这一章将要介绍的范例更接近于生产部署配置。该章内容更深一些，也将促使你思考今后该如何设计更复杂的微服务。

你所需要了解的技术

这是一本中等难度的书，要求读者对一些开发和系统管理概念有最基本的理解。在深入学习本书之前，你可能需要了解以下内容。

- bash（Unix shell）

  这是 Linux 和 OS X 上默认的 Unix shell。读者最好熟悉 Unix shell，如编辑文件、设置文件权限、在文件系统中移动文件、管理用户权限，以及一些基本的 shell 编程知识。如果你对 Linux shell 不太熟悉，可以参考一下 Cameron Newham 的 *Learning the Bash Shell*，或者 Carl Albing、JP Vossen 和 Cameron Newham 合著的 *Bash Cookbook*，这两本书也都是由 O’Reilly 出版的。

- 软件包管理

  本书中的工具往往需要通过安装一些软件包来满足多个依赖。因此这就要求读者对所用计算机系统上的软件包管理程序有所了解。这可能是 Ubuntu/Debian 系统上的 apt，或是 CentOS/RHEL 系统上的 yum，又或是 OS X 上的 port 或者 brew。不管你使用的是什么软件包管理器，请确保你知道如何安装、升级和删除软件包。

- Git

  Git 已成为分布式版本控制领域的标准。如果你已经熟悉 CVS 和 SVN，但还没有使用过 Git，那你真应该开始使用 Git。Jon Loeliger 和 Matthew McCullough 合著的《Git 版本控制管理（第 2 版）》是很好的入门教材。如果你想使用 Git，还想托管自己的代码仓库，那么 GitHub 网站（http://github.com/）是一个很好的资源。要想学习 GitHub，可以访问 http://training.github.com 以及相关的交互式教程（http://try.github.io/）。

- Python

  除了 C/C++ 或 Java 编程语言，我总是鼓励学生选择一门脚本语言。Perl 曾经统治了世界，然而现如今，Ruby 和 Go 语言则更加普及。我本人使用 Python，本书中的大多数例子将使用 Python 来编写，但也有几个例子使用了 Ruby，甚至还有一个例子使用了 Clojure。O’Reilly 出版了很多 Python 方面的图书，包括 Bill Lubanovic 的《Python 语言及其应用》[1]、Mark Lutz 的《Python 编程》，以及 David Beazley 和 Brian K. Jones 合著的《Python Cookbook 中文版》。

- Vagrant

  Vagrant 已经成为 DevOps 工程师建立和管理虚拟环境的优秀工具之一。它非常适合用

注 1：此书已由人民邮电出版社出版。——编者注

于在本地测试和快速配置虚拟机，但我们也可以使用一些插件来通过 Vagrant 连接到公共云提供商。本书将采用 Vagrant 来快速部署一个虚拟机实例，使其像 Docker 主机一样运行。你也可能想阅读一下由 Vagrant 的开发者 Mitchell Hashimoto 本人写的 *Vagrant: Up and Running* 一书。

- Go

  Docker 采用 Go 语言编写。在过去的几年里，Go 语言已成为许多初创公司首选的新编程语言。本书不是教大家如何学习 Go 编程的，而是会介绍如何编译一些 Go 项目。我希望读者至少知道如何安装一个 Go 工作区。如果想了解更多，Introduction to Go Programming（http://shop.oreilly.com/product/0636920035305.do）这个视频培训课程是一个很好的选择。

在线内容

本书中使用的示例代码、Vagrantfile 和其他脚本都可以从 GitHub 找到（https://github.com/how2dock/docbook）。你可以克隆这个仓库，进入相应章节和范例并使用其中的代码。比如，下面的代码显示了如何通过 Vagrant 启动一台 Ubuntu 14.04 虚拟机并在其中安装 Docker。

```
$ git clone https://github.com/how2dock/docbook.git
$ cd dockbook/ch01/ubuntu14.04/
$ vagrant up
```

> 这个仓库中的示例都不是最优配置，仅足以运行范例中的示例。

排版约定

本书使用了下列排版约定。

- 楷体

  表示新术语。

- 等宽字体（`constant width`）

  表示程序片段，以及正文中出现的变量、函数名、数据库、数据类型、环境变量、语句和关键字等。

- 加粗等宽字体（**`constant width bold`**）

  表示应该由用户输入的命令或其他文本。

- 等宽斜体（*`constant width italic`*）

  表示应该由用户输入的值或根据上下文确定的值替换的文本。



该图标表示提示或建议。



该图标表示一般注记。



该图标表示警告或警示。

Safari® Books Online

Safari Books Online（http://www.safaribooksonline.com）是应运而生的数字图书馆。它同时以图书和视频的形式出版世界顶级技术和商务作家的专业作品。技术专家、软件开发人员、Web 设计师、商务人士和创意专家等，在开展调研、解决问题、学习和认证培训时，都将 Safari Books Online 视作获取资料的首选渠道。

对于组织团体、政府机构和个人，Safari Books Online 提供各种产品组合和灵活的定价策略。用户可通过一个功能完备的数据库检索系统访问 O'Reilly Media、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology 以及其他几十家出版社的上千种图书、培训视频和正式出版之前的书稿。要了解 Safari Books Online 的更多信息，我们网上见。

联系我们

请把对本书的评价和问题发给出版社。

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

O’Reilly 的每一本书都有专属网页，你可以在那儿找到本书的相关信息，包括勘误表、示例代码以及其他信息。本书的网站地址是：

http://shop.oreilly.com/product/0636920036791.do

对于本书的评论和技术性问题，请发送电子邮件到：

bookquestions@oreilly.com

要了解更多 O’Reilly 图书、培训课程、会议和新闻的信息，请访问以下网站：

http://www.oreilly.com

我们在 Facebook 的地址：http://facebook.com/oreilly

请关注我们的 Twitter 动态：http://twitter.com/oreillymedia

我们的 YouTube 视频地址：http://www.youtube.com/oreillymedia

致谢

我写这本书用了 8 个月的时间。在这段时间里，我阅读了无数关于 Docker 的博客和文档，并对一切进行了反复的测试。我要感谢我的妻子和孩子，他们允许我利用周末和晚上的时间来写作这本书。我还要感谢 Brian Anderson，他一直督促我、鼓励我，多亏他的定期检查，我才能准时完成目标。如果没有 Fintan Ryan、Eugene Yakubovich、Joe Beda 和 Pini Riznik 的补充，这本书也不会这么完整。他们四个人帮我完成了非常有价值的内容，这也将会帮到许多读者。非常感谢 Patrick Debois 和 John Willis 的早期审阅，他们提供了令人鼓舞的宝贵的反馈，这也使得这本书更完善。Ksenia Burlachenko 和 Carlos Sanchez 的仔细审阅帮助我解决了很多问题，这对所有读者都非常有帮助。非常感谢他们两个。特别感谢 Funs Kessen，他是一位网络技术和应用程序设计专家，从来没有拒绝回答我提出的许多愚蠢问题。最后，非常感谢本书早期发布版的读者，特别是 Olivier Boudry，感谢他们愿意阅读内容尚不完整，并且存在拼写错误、语法错误和内容错误的版本。没有他们的更正和评论，这本书也不会像现在这样好。

电子书

扫描如下二维码，即可购买本书电子版。

第 1 章
Docker入门

1.0 简介

入门 Docker 很简单。Docker 的核心是一个被称作 Docker 引擎的基于单主机运行的守护进程，我们可以通过这个守护进程来创建和管理容器。在深入使用 Docker 之前，你需要先在一台主机（比如台式机、笔记本电脑或者服务器）上安装 Docker 引擎。

本章中的前几个范例将会介绍在服务器上运行 Docker 所需的安装步骤。官方文档差不多涵盖了所有操作系统，这里我们会对 Ubuntu 14.04（参见范例 1.1）、CentOS 6.5（参见范例 1.2）和 CentOS 7（参见范例 1.3）进行介绍。如果你想使用 Vagrant，可以参见范例 1.4。

我们也会以树莓派为例来介绍如何在 ARM CPU 上安装 Docker（参见范例 1.5）。如果是 Windows 或者 OS X 系统的主机，你可以使用 Docker Toolbox（参见范例 1.6）。Docker Toolbox 除了包括 Docker 引擎之外，还包括其他一些实用工具。Docker Toolbox 使用 VirtualBox 将一个虚拟机作为 Docker 主机运行，这个虚拟机也就是 Boot2Docker。现在已经不再推荐使用 Boot2Docker 了，不过我们还是会在范例 1.7 中介绍一下如何使用 Boot2Docker 安装 Docker。

在这些安装范例之后，我们会介绍一下 `docker-machine`。这是一个 Docker 工具，可以通过它在公有云上创建云主机并安装 Docker，自动配置本地 Docker 客户端来使用远程 Docker 主机。范例 1.9 中会介绍如何在 DigitalOcean 云中使用 `docker-machine`。

一旦在自己的环境中安装好了 Docker，就可以浏览一下创建和管理容器所需的基本命令。范例 1.11 将会展示运行容器的第一步，范例 1.13 将会带你了解一个容器的标准生命周期：创建、启动、停止、终止和移除。

介绍完这些基本概念之后，我们将会直接深入到 Dockerfile（参见范例 1.14）。Dockerfile 是一个用于描述如何构建容器镜像的文件。它是 Docker 中的一个核心概念，第 2 章会详细展开讨论，这里只介绍其最简单的用法。我们通过 Dockerfile 来学习一个更为复杂的例子，即运行 WordPress 服务。

首先，我们会从头开始构建一个 Docker 镜像，在单一容器中运行多个进程（参见范例 1.15）。Docker 会改变我们的应用程序设计思想，不再将所有一切打包在一起，而是创建多个独立的服务，然后将这些独立的服务连接起来。然而，这并不意味着一个容器中不能运行多个服务。使用 supervisord 可以在一个容器中运行多个服务，范例 1.15 将会讲述如何去做。但是 Docker 的强大之处在于通过组合服务来运行应用程序。因此，在范例 1.16 中，我们会介绍如何将单一容器示例拆分为两个容器，并使用容器链接进行互连。这将是你的第一个分布式应用程序，尽管它也只是运行在同一台主机之上。

我们在本章介绍的最后一个概念是数据管理。在容器中访问数据是一个关键组件。你可以通过它来加载配置变量或数据集，或在容器之间共享数据。我们将再次使用 WordPress 的例子，告诉你如何备份数据库（参见范例 1.17），如何将宿主机的数据挂载到容器中（参见范例 1.18），以及如何创建数据容器（参见范例 1.19）。

总之，在本章中，你将会快速学到如何在一台主机上安装 Docker 引擎，并运行一个由两个容器组成的 WordPress 网站。

1.1 在Ubuntu 14.04上安装Docker

1.1.1 问题

你想在 Ubuntu 14.04 上运行 Docker。

1.1.2 解决方案

在 Ubuntu 14.04 上，可以通过至多三条 bash 命令来安装 Docker。Docker 项目推荐的安装方式是从网上下载并运行一个 bash 脚本。需要注意的是，在 Ubuntu 的软件包仓库中已经有一个 docker 软件包，不过这个软件与 Docker（http://www.docker.com）并没有任何关系。执行推荐的安装，如下所示。

```
$ sudo apt-get update
$ sudo apt-get install -y wget
$ sudo wget -qO- https://get.docker.com/ | sh
```

可以通过查看 Docker 软件的版本来确认是否已正确安装了 Docker，如下所示。

```
$ sudo docker --version
Docker version 1.7.1, build 786b29d
```

可以停止、启动和重启 Docker 服务。比如，可以像下面这样重启 Docker 服务。

```
$ sudo service docker restart
```

如果你想直接以一个非 root 用户的身份来运行 docker 命令，可以将该用户添加到 docker 用户组，如下所示。

```
$ sudo gpasswd -a <user> docker
```

退出当前 shell 然后重新登录，或者重新启动一个新 shell，就能使用上面的配置了。

1.1.3 讨论

你可以按照 https://get.docker.com 的安装脚本，一步一步地手动安装或者进行自定义安装。在 Ubuntu 14.04（代号 trusty）上，最精简的安装步骤如下所示。

```
$ sudo apt-get update
$ sudo apt-get install -y linux-image-extra-$(uname -r) linux-image-extra-virtual
$ sudo apt-key adv --keyserver hkp://p80.pool.sks-keyservers.net:80
                   --recv-keys 58118E89F3A912897C070ADBF76221572C52609D
$ sudo su
# echo deb https://apt.dockerproject.org/repo ubuntu-trusty main > \
/etc/apt/sources.list.d/docker.list
# apt-get -y install docker-engine
```

1.1.4 参考

- 如果你想在其他操作系统上安装 Docker，请参考官方的安装文档（https://docs.docker.com/docker/installation/）。

1.2 在CentOS 6.5上安装Docker

1.2.1 问题

你想在 CentOS 6.5 上安装 Docker。

1.2.2 解决方案

在 CentOS 6.5 上，可以通过添加 EPEL（Extra Packages for Enterprise Linux）仓库使用 docker-io 包安装 Docker，如下所示。

```
$ sudo yum -y update
$ sudo yum -y install epel-release
$ sudo yum -y install docker-io
$ sudo service docker start
$ sudo chkconfig docker on
```

在 CentOS 6.5 上，Docker 的版本应该是 1.6.2，如下所示。

```
# docker --version
Docker version 1.6.2, build 7c8fca2/1.6.2
```

1.2.3　讨论

Docker 将不会继续提供对 CentOS 6.*x* 的支持。如果你想使用最新版 Docker，那么应该选择 CentOS 7（参见范例 1.3）。

1.3　在CentOS 7上安装Docker

1.3.1　问题

你想在 CentOS 7 上使用 Docker。

1.3.2　解决方案

通过 yum 管理器安装 Docker 软件包。CentOS 采用了 systemd，因此你需要使用 `systemctl` 命令来管理 docker 服务，如下所示。

```
$ sudo yum update
$ sudo yum -y install docker
$ sudo systemctl start docker
```

也可以使用 Docker 官方的安装脚本来安装，这也会使用 Docker 仓库中的软件包，如下所示。

```
$ sudo yum update
$ sudo curl -sSL https://get.docker.com/ | sh
```

1.4　使用Vagrant创建本地Docker主机

1.4.1　问题

你的本地计算机的操作系统和你想要运行 Docker 的操作系统不同。比如，你现在运行的是 OS X，但是你想在 Ubuntu 上运行 Docker。

1.4.2　解决方案

在本地通过 Vagrant（http://vagrantup.com）启动一个虚拟机（virtual machine，VM），并使用 Vagrantfile 中的 shell 配置程序初始化 VM。

如果已经安装了 VirtualBox（http://virtualbox.org）和 Vagrant（http://vagrantup.com），那么你只需要创建一个名为 Vagrantfile 的文本文件，其内容如下所示。

```
VAGRANTFILE_API_VERSION = "2"

$bootstrap=<<SCRIPT
apt-get update
apt-get -y install wget
wget -qO- https://get.docker.com/ | sh
gpasswd -a vagrant docker
```

```
service docker restart
SCRIPT

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
  config.vm.box = "ubuntu/trusty64"

  config.vm.network "private_network", ip: "192.168.33.10"

  config.vm.provider "virtualbox" do |vb|
     vb.customize ["modifyvm", :id, "--memory", "1024"]
  end

  config.vm.provision :shell, inline: $bootstrap

end
```

之后就可以启动虚拟机了。Vagrant 将会从 Vagrant cloud［http://vagrantcloud.com，现在是 Atlas（https://atlas.hashicorp.com）的一部分］下载 ubuntu/trusty64 镜像（box），通过 VirtualBox 创建该镜像的一个实例，再运行在 Vagrantfile 中定义的初始化脚本。这个虚拟机实例将会有 1GB 的内存和两个网络接口：一个用于与外部网络进行通信的*网络地址转换*（Network Address Translation，NAT）接口，一个本地网络接口 192.168.33.10。虚拟机启动后，就可以通过 `ssh` 来登录到虚拟机并使用 Docker。

```
$ vagrant up
$ vagrant ssh
vagrant@vagrant-ubuntu-trusty-64:~$ docker ps
CONTAINER ID    IMAGE     COMMAND     CREATED     STATUS     PORTS     NAMES
```

在上面的 Vagrant 配置中，vagrant 用户被添加到了 Docker 用户组。这样一来，即使你不是 root 用户，也能运行 Docker 命令。你可以在 how2doc 仓库（https://github.com/how2dock/docbook.git）的 ch01 文件夹中找到上面的脚本。

1.4.3 讨论

如果从未使用过 Vagrant，你需要先安装它。Vagrant 官方下载页面（https://www.vagrantup.com/downloads）上列出了主流的安装包类型。比如对于基于 Debian 的系统，可以选择下载 .deb 包，并像下面这样安装。

```
$ wget https://dl.bintray.com/mitchellh/vagrant/vagrant_1.7.4_x86_64.deb
$ sudo dpkg -i vagrant_1.7.4_x86_64.deb
$ sudo vagrant --version
Vagrant 1.7.4
```

1.5 在树莓派上安装Docker

1.5.1 问题

假定你在公司大量使用树莓派（https://www.raspberrypi.org），或者作为个人兴趣在业余时间喜欢自己钻研一下。不管是哪种情况，你都想知道如何在自己的树莓派上安装 Docker。

1.5.2 解决方案

你可以从 Hypriot（http://blog.hypriot.com）下载预配置好的 SD 卡镜像。你需要按照下面的步骤来操作。

(1) 下载 SD 卡镜像（http://blog.hypriot.com/downloads）。
(2) 将镜像写入 SD 卡。
(3) 将 SD 卡插入树莓派并启动。
(4) 登录到树莓派，开始使用 Docker。

1.5.3 讨论

比如在一台 OS X 主机上，你可以按照 Hypriot 的指南（http://blog.hypriot.com/getting-started-with-docker-and-mac-on-the-raspberry-pi/）来操作。

下载并解压 SD 镜像，如下所示。

```
$ curl -sOL http://downloads.hypriot.com/hypriot-rpi-20150416-201537.img.zip
$ unzip hypriot-rpi-20150416-201537.img.zip
```

然后将 SD 卡插入到主机上的读卡器中，查看所有可用磁盘，找到哪个是你所使用的 SD 卡。之后你需要卸载这个磁盘，并使用 `dd` 命令将镜像复制到这张 SD 卡上。这里我们假定 SD 卡设备名为 `disk1`。

```
$ diskutil list
...
$ diskutil unmountdisk /dev/disk1
$ sudo dd if=hypriot-rpi-20150416-201537.img of=/dev/rdisk1 bs=1m
$ diskutil unmountdisk /dev/disk1
```

将镜像复制到 SD 卡之后，对 SD 卡执行弹出操作，将 SD 卡从读卡器中移除，然后插入到树莓派中。你需要知道树莓派的 IP 地址，这样就可以通过 `ssh` 使用密码 `hypriot` 登录到树莓派了，如下所示。

```
$ ssh root@<IP_OF_RPI>
...
HypriotOS: root@black-pearl in ~
$ docker ps
CONTAINER ID     IMAGE     COMMAND     CREATED     STATUS     PORTS     NAMES
HypriotOS: root@black-pearl in ~
$ uname -a
```

```
Linux black-pearl 3.18.11-hypriotos-v7+ #2 SMP PREEMPT Sun Apr 12 16:34:20 UTC \
2015 armv7l GNU/Linux
HypriotOS: root@black-pearl in ~
```

你已经拥有了一个可以在 ARM 系统上运行的 Docker 了。

因为容器会使用 Docker 主机的内核，所以你需要拉取为基于 ARM 的架构准备的镜像。访问 Docker Hub 可以查找为基于 ARM 的系统准备的镜像，选择一个 Hyperiot 提供的镜像是不错的开始。

1.5.4 参考

- 在 Windows 上为树莓派安装 Docker 的指令（http://blog.hypriot.com/getting-started-with-docker-and-windows-on-the-raspberry-pi/）
- 在 Linux 上为树莓派安装 Docker 的指令（http://blog.hypriot.com/getting-started-with-docker-and-linux-on-the-raspberry-pi/）

1.6 在OS X上通过Docker Toolbox安装Docker

1.6.1 问题

Docker 守护进程并不支持 OS X 系统，但是你想在 OS X 上使用 Docker。

1.6.2 解决方案

可以使用 Docker Toolbox（https://https://www.docker.com/toolbox），这是一个包含了 Docker 客户端、Docker Machine、Docker Compose、Docker Kitematic 和 VirtualBox 的安装包。Docker Toolbox 允许你在 VirtualBox 中启动一个运行着 Docker 守护进程的微型虚拟机。安装到你的 OS X 操作系统上的 Docker 客户端也会被配置为连接到这个虚拟机中的 Docker 守护进程。安装 Docker Toolbox 的同时也会安装 Docker Machine（参见范例 1.9）、Docker Compose（参见范例 7.1）和 Kitematic（参见范例 7.5）。

你可以从 Docker Toolbox 的下载页面（https://https://www.docker.com/toolbox）下载它的安装包。下载完成之后（参见图 1-1），打开安装文件并按照提示步骤进行安装即可。安装完成之后，finder 应用会自动打开，你将会看到一个指向 Docker quickstart terminal 的链接。单击这个链接会打开一个终端，同时会自动在 VirtualBox 中启动一个虚拟机。Docker 客户端也会自动被配置为连接到在这个虚拟机中运行的 Docker 守护进程。

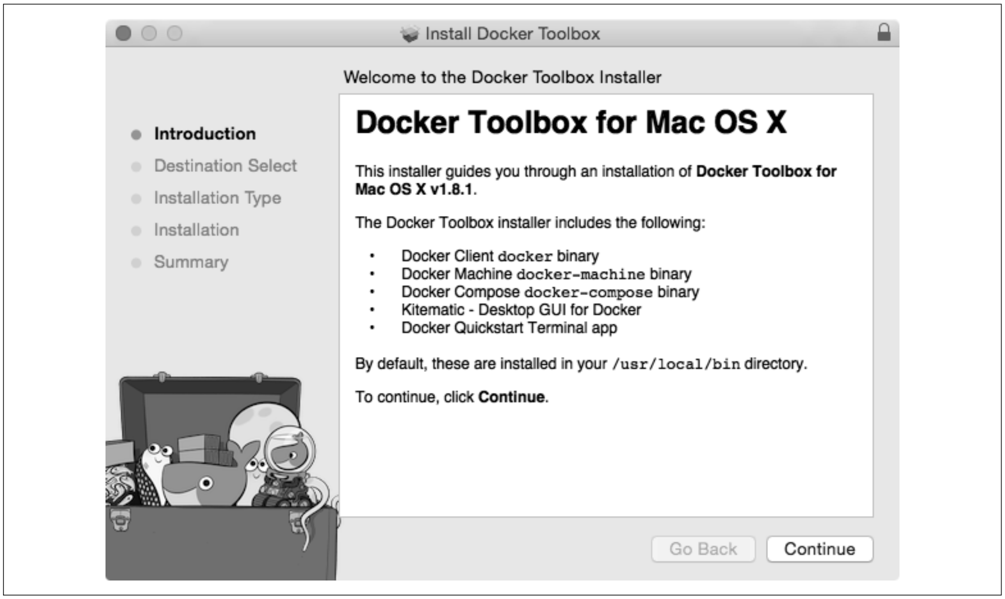


图 1-1：在 OS X 上的 Docker Toolbox

Toolbox 终端显示 Docker 客户端被配置为使用默认虚拟机。可以像下面这样试用一些 `docker` 命令。

```
                        ##         .
                  ## ## ##        ==
               ## ## ## ## ##    ===
           /”””””””””””””””””\___/ ===
      ~~~ {~~ ~~~~ ~~~ ~~~~ ~~~ ~ /  ===- ~~~
           \______ o           __/
             \    \         __/
              \____\_______/


docker is configured to use the default machine with IP 192.168.99.100
For help getting started, check out the docs at https://docs.docker.com

bash-4.3$ docker ps
CONTAINER ID        IMAGE    COMMAND     CREATED     STATUS      PORTS       NAMES
bash-4.3$ docker images
REPOSITORY          TAG      IMAGE ID    CREATED     VIRTUAL SIZE
bash-4.3$ docker
docker          docker-compose  docker-machine
```

可以看到，现在你也可以使用 `docker-machine` 和 `docker-compose` 二进制文件了。

1.6.3 讨论

Docker Toolbox 是从 Docker 1.8 开始提供的，并且在 OS X 和 Windows 主机上，你也应该使用它作为默认的安装方式。在 Docker Toolbox 之前的版本中，你可以使用 Boot2Docker（参见范例 1.7 和范例 1.8）。虽然本书中也为 Boot2Docker 准备了一些范例，但是你还是应该优先使用 Docker Toolbox。实际上，Boot2Docker 仍然用在 Toolbox 中。你可以像下面这样通过 `docker-machine ssh` 登录到 VM 来确认这一点。

```
bash-4.3$ docker-machine ssh default
                       ##         .
                 ## ## ##        ==
              ## ## ## ## ##    ===
          /”””””””””””””””””\___/ ===
     ~~~ {~~ ~~~~ ~~~ ~~~~ ~~~ ~ /  ===- ~~~
          \______ o           __/
            \    \         __/
             \____\_______/
_                 _   ____     _            _
| |__   ___   ___ | |_|___ \ __| | ___   ___| | _____ _ __
| ‘_ \ / _ \ / _ \| __| __) / _` |/ _ \ / __| |/ / _ \ ‘__|
| |_) | (_) | (_) | |_ / __/ (_| | (_) | (__|   <  __/ |
|_.__/ \___/ \___/ \__|_____\__,_|\___/ \___|_|\_\___|_|
Boot2Docker version 1.8.1, build master : 7f12e95 - Thu Aug 13 03:24:56 UTC 2015
Docker version 1.8.1, build d12ea79
```

1.7 在OS X上通过Boot2Docker安装Docker

1.7.1 问题

Docker 守护进程并不支持 OS X 系统，但是你想在 OS X 上无缝地使用 Docker。

1.7.2 解决方案

使用轻量 Linux 发行版 Boot2Docker（http://boot2docker.io）。Boot2Docker 基于微内核 Linux，并且为运行 Docker 进行了专门配置。安装完之后，你会得到一个 `boot2docker` 命令。你将会使用这个命令来与通过 VirtualBox 启动的虚拟机进行交互，这个虚拟机将会作为 Docker 主机。运行于 OS X 之上的 Docker 客户端和 Docker 守护进程不一样，需要在本地的 OS X 上进行安装。

让我们这就开始下载并安装 Boot2Docker。打开 Boot2Docker 网站（http://boot2docker.io）将会看到一些下载链接。从它的发布页面中（https://github.com/boot2docker/osx-installer/releases）选择最新的版本下载。下载完成之后，打开安装文件，如图 1-2 所示。

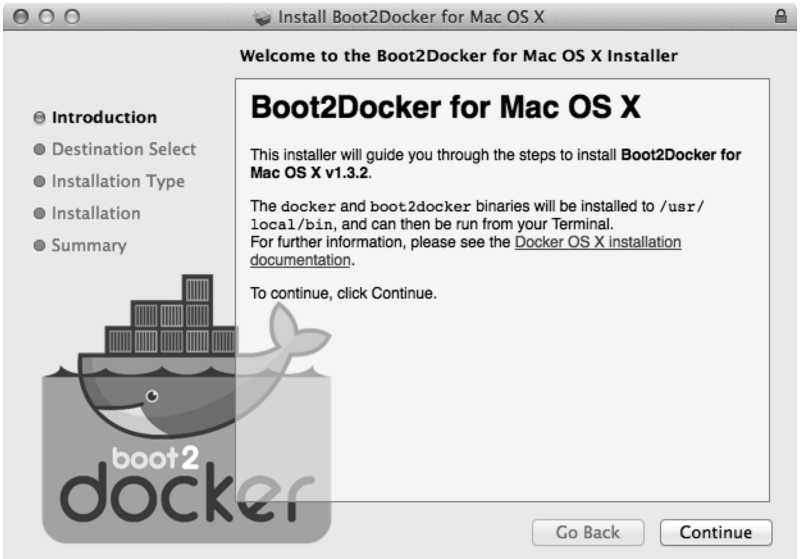


图 1-2：Boot2Docker 安装向导

安装完成后（参见图 1-3），你就可以开始使用 Boot2Docker 了。



图 1-3：Boot2Docker 安装完毕

终端窗口中，在出现提示时键入 `boot2docker`，此时应该能看到这个命令的用法选项。也可以查看所安装的 Boot2Docker 版本号，如下所示。

```
$ boot2docker
Usage: boot2docker [<options>] {help|init|up|ssh|save|down|poweroff|reset|
                               restart|config|status|info|ip|shellinit|delete|
                               download|upgrade|version} [<args>]
$ boot2docker version
Boot2Docker-cli version: v1.3.2
Git commit: e41a9ae
```

在安装完 Boot2Docker 之后，第一步是需要对它进行初始化。如果还没有下载 Boot2Docker 的 ISO 镜像，那么这一步将会下载这个镜像并在 VirtualBox 中创建一个虚拟机，如下所示。

```
$ boot2docker init
Latest release for boot2docker/boot2docker is v1.3.2
Downloading boot2docker ISO image...
Success:
    downloaded https://github.com/boot2docker/boot2docker/releases/download/\
  v1.3.2/boot2docker.iso
    to /Users/sebgoa/.boot2docker/boot2docker.iso
```

可以看到，ISO 镜像被保存到了用户主文件夹下的 .boot2docker/boot2docker.iso 文件夹中。如果打开 VirtualBox 界面，将会看到 boot2docker 这个虚拟主机处于关机状态（参见图 1-4）。



图 1-4：boot2docker VirualBox 虚拟机

你并不需要保持 VirtualBox 界面为打开状态，上面的截图只是为了方便说明。Boot2Docker 会在后台运行，使用 VBoxManage 命令对 boot2docker 虚拟机进行管理。

你现在已经可以使用 Boot2Docker 了。下面的命令将会启动这个虚拟机，然后返回一些操作指令，你可以按照这些指令对环境变量进行设置，以正确地连接到虚拟机中的 Docker 守护进程。

```
$ boot2docker start
Waiting for VM and Docker daemon to start...
.........................ooooooooooooooooooooo
Started.
Writing /Users/sebgoa/.boot2docker/certs/boot2docker-vm/ca.pem
Writing /Users/sebgoa/.boot2docker/certs/boot2docker-vm/cert.pem
Writing /Users/sebgoa/.boot2docker/certs/boot2docker-vm/key.pem

To connect the Docker client to the Docker daemon, please set:
    export DOCKER_CERT_PATH=/Users/sebgoa/.boot2docker/certs/boot2docker-vm
    export DOCKER_TLS_VERIFY=1
    export DOCKER_HOST=tcp://192.168.59.103:2376
```

虽然我们可以自己手动来设置这些环境变量，但是 Boot2Docker 提供了一个方便的命令：`shellinit`。可以使用该命令设置连接到 Docker 守护进程所需要的传输层安全（Transport Layer Security，TLS）信息，之后就可以在本地 OS X 计算机上访问虚拟机中的 Docker 主机了，如下所示。

```
$ $(boot2docker shellinit)
Writing /Users/sebgoa/.boot2docker/certs/boot2docker-vm/ca.pem
Writing /Users/sebgoa/.boot2docker/certs/boot2docker-vm/cert.pem
Writing /Users/sebgoa/.boot2docker/certs/boot2docker-vm/key.pem
$ docker ps
CONTAINER ID    IMAGE    COMMAND    CREATED    STATUS    PORTS    NAMES
```

1.7.3 讨论

当有新版本的 Boot2Docker 可用时，你非常容易就能升级。只需要使用 `download` 命令下载最新的 Boot2Docker 安装包和新的 ISO 镜像就可以了。

请确保通过 `$ boot2docker stop` 命令停止 boot2docker 虚拟机之后，再运行安装脚本：

```
$ boot2docker stop
$ boot2docker upgrade
$ boot2docker start
```

1.8 在Windows 8.1台式机上运行Boot2Docker

1.8.1 问题

你有一台安装了 Windows 8.1 的台式机，你想在这台计算机上使用 Boot2Docker 来测试一下 Docker。

1.8.2 解决方案

使用 Boot2Docker 的 Windows 安装程序（https://github.com/boot2docker/windows-installer/releases/tag/v1.8.0），参见图 1-5。

下载完最新版本的 Windows 安装程序（一个 .exe 二进制文件）之后，通过命令提示符或者资源管理器（参见图 1-5）运行这个安装程序。这将会自动安装 VirtualBox、MSysGit 和 Boot2Docker 的 ISO 镜像。要想在 Windows 主机上使用 ssk-keygen 二进制文件，需要 MSysGit 这个软件。按照安装向导的提示进行安装，在此过程中，你需要同意一些来自 Oracle 关于 VirtualBox 的许可证信息。这个安装程序会在桌面上创建 VirtualBox 和启动 Boot2Docker 的快捷方式。



图 1-5：Boot2Docker Windows 8.1 安装程序

当安装完成之后，双击 Boot2Docker 的快捷方式，就可以在 VirtualBox 中启动虚拟机，并

打开一个命令提示符（参见图 1-6）。这时就可以开始在 Windows 主机中使用 Docker 了。

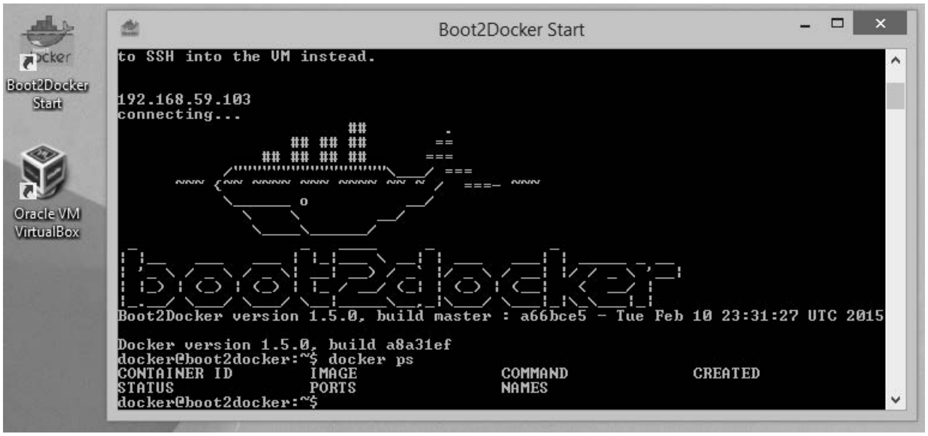


图 1-6：Boot2Docker Windows 8.1 命令

1.8.3　讨论

Docker Machine（参见范例 1.9）也自带了一个 Hyper-V 驱动程序。如果你的计算机上已安装 Hyper-V，那么也可以通过 Docker Machine 来启动 Boot2Docker 实例。

1.8.4　参考

- 关于如何在 Windows 上使用 Boot2Docker 的 Docker 官方文档（https://docs.docker.com/installation/windows/）

1.9　使用Docker Machine在云中创建Docker主机

1.9.1　问题

你不想在本地通过 Vagrant 来安装 Docker 守护进程（参见范例 1.4），也不想使用 Boot2Docker（参见范例 1.7）。事实上，你想在云上使用 Docker 主机（比如 AWS、DigitalOcean、Azure 或 Google Compute Engine），并从本地 Docker 客户端无缝地连接到云上的 Docker 主机。

1.9.2　解决方案

使用 Docker Machine 在你选择的公有云上启动一个云主机实例。Docker Machine 是一个在本地主机上运行的客户端工具，你可以使用这个工具在远端的公有云中启动一台服务器，并像使用本地 Docker 主机一样来使用远程 Docker 主机。Machine 将会自动安装 Docker 并配置使用 TLS 进行安全传输。这之后就可以将远程的云主机实例作为 Docker 主机，从本

地 Docker 客户端进行操作。可以参考第 8 章，那里有更多专门为在云中使用 Docker 而准备的范例。

> Docker Machine 的 beta 版是在 2015 年 2 月 26 日发布的（http://blog.docker.com/2015/02/announcing-docker-machine-beta/），官方文档（https://docs.docker.com/machine/）可以在 Docker 的网站上看到。它的源代码也在 GitHub 上（https://github.com/docker/machine）。

让我们这就开始吧。Machine 现在支持 VirtualBox、DigitalOcean（https://www.digitalocean.com）、Amazon Web Services（https://aws.amazon.com）、Azure（https://azure.microsoft.com）、Google Compute Engine（GCE，http://cloud.google.com）以及其他一些云服务提供商。也有一些驱动程序正在开发或审查中，你可以从这里（https://github.com/docker/machine/pulls）查看详细情况，我们可以期待很快会有更多的驱动程序出现。本范例将使用 DigitalOcean，如果你也想一步一步跟着操作，需要先到 DigitalOcean（https://cloud.digitalocean.com/registrations/new）注册一个账号。

账号注册完成之后，先不要通过 DigitalOcean 的界面来创建云主机（droplet）。取而代之的是，创建一个 API 访问令牌，以供 Docker Machine 使用。这个令牌是一个同时具备 read 和 write 权限的令牌，这样 Machine 才能上传 SSH 公钥（图 1-7）。之后，在本地电脑的命令行中设置一个名为 `DIGITALOCEAN_ACCESS_TOKEN` 的环境变量，其值即为刚才创建的令牌的内容。

> Machine 会将 SSH 公钥上传到你的云账号上。请确认你的访问令牌或 API 密钥具有创建公钥的必要权限。

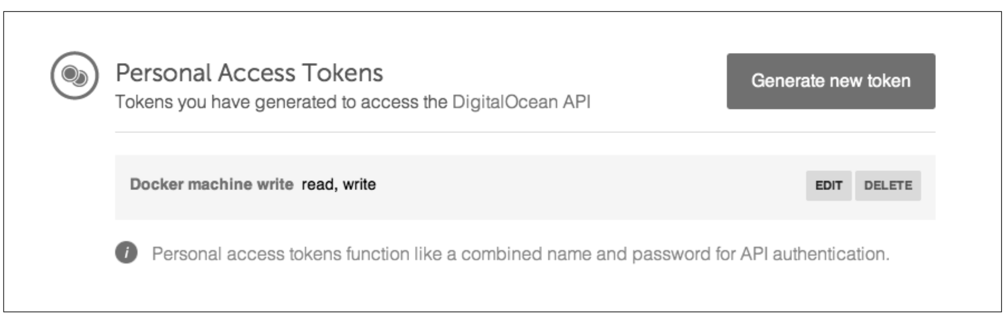


图 1-7：DigitalOcean 为 Machine 准备的访问令牌

一切准备就绪。你现在需要去下载 docker-machine 二进制文件。 打开 Docker Machine 文档网站（https://docs.docker.com/machine/），并选择与本地主机架构一致的二进制文件。比如，在 OS X 上像下面这样操作。

```
$ curl -sOL https://github.com/docker/machine/releases/download/v0.3.0/ \
docker-machine_darwin-amd64
```

```
$ mv docker-machine_darwin-amd64 docker-machine
$ chmod +x docker-machine
$ ./docker-machine --version
docker-machine version 0.3.0
```

由于已经设置了 DIGITALOCEAN_ACCESS_TOKEN 环境变量，现在就可以创建远程 Docker 主机了，如下所示。

```
$ ./docker-machine create -d digitalocean foobar
Creating SSH key...
Creating Digital Ocean droplet...
To see how to connect Docker to this machine, run: docker-machine env foobar
```

回到 DigitalOcean 的控制面板，你将会看到一个新创建的 SSH 密钥，以及一个新创建的云主机（参见图 1-8 和图 1-9）。



图 1-8：DigitalOcean 上由 Machine 生成的 SSH 密钥

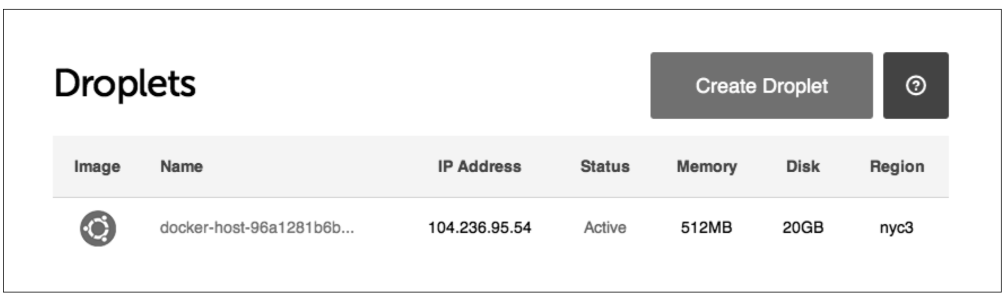


图 1-9：由 Machine 创建的 DigitalOcean 云主机

要想将本地 Docker 客户端配置为使用这台远程 Docker 主机，可以执行下面的命令，根据该命令的输出来进行设置。

```
$ ./docker-machine env foobar
export DOCKER_TLS_VERIFY="1"
export DOCKER_HOST="tcp://104.131.92.15:2376"
export DOCKER_CERT_PATH="/Users/sebastiengoasguen/.docker/machine/machines/foobar"
export DOCKER_MACHINE_NAME="foobar"
# Run this command to configure your shell:
# eval "$(docker-machine env foobar)"
$ eval "$(./docker-machine env foobar)"
$ docker ps
CONTAINER ID    IMAGE    COMMAND    CREATED    STATUS    PORTS    NAMES
```

现在就可以享受一个由 Docker Machine 创建的位于远程 DigitalOcean 主机上的 Docker 了。

1.9.3 讨论

> 如果在命令行上没有指定参数，Machine 就会去查找 `DIGITALOCEAN_IMAGE`、`DIGITALOCEAN_REGION` 和 `DIGITALOCEAN_SIZE` 等环境变量。默认情况下，这些环境变量会被分别设置为 docker、nyc3 和 512mb。

docker-machine 二进制文件允许你在多个提供程序上创建多台主机。它也提供了一些基本的管理功能，比如 `start`、`stop` 和 `rm` 等，如下所示。

```
$ ./docker-machine
...
COMMANDS:
   active Get or set the active machine
   create Create a machine
   config Print the connection config for machine
   inspect  Inspect information about a machine
   ip   Get the IP address of a machine
   kill   Kill a machine
   ls   List machines
   restart  Restart a machine
   rm   Remove a machine
   env    Display the commands to set up the environment for the Docker client
   ssh    Log into or run a command on a machine with SSH
   start  Start a machine
   stop   Stop a machine
   upgrade  Upgrade a machine to the latest version of Docker
   url    Get the URL of a machine
   help, h  Shows a list of commands or help for one command
```

比如，可以列出刚才创建的主机，获得该主机的 IP 地址，甚至可以通过 SSH 连接到该主机，如下所示。

```
$ ./docker-machine ls
NAME     ACTIVE   DRIVER         STATE     URL                         SWARM
foobar   *        digitalocean   Running   tcp://104.131.92.15:2376
$ ./docker-machine ip foobar
104.131.92.15
$ ./docker-machine ssh foobar
Welcome to Ubuntu 14.04.2 LTS (GNU/Linux 3.13.0-57-generic x86_64)
...

Last login: Mon Mar 16 09:02:13 2015 from ...
root@foobar:~#
```

在本范例结束之前，别忘了删除刚才创建的云主机，如下所示。

```
$ ./docker-machine rm foobar
```

1.9.4　参考

- Docker Machine 官方文档（https://docs.docker.com/machine/）

1.10　使用Docker实验版二进制文件

1.10.1　问题

你想使用 Docker 的实验性功能，或者想使用提交给 Docker 上游代码的补丁。

1.10.2　解决方案

使用 Docker 实验版二进制文件。你可以下载实验版二进制文件或者使用每天晚上更新的实验版通道。

以 Linux 发行版中软件包的形式获取 Docker 的每日构建，如下所示。

```
$ wget -qO- https://experimental.docker.com/ | sh
$ docker version | grep Version
 Version:      1.8.0-dev
 Version:      1.8.0-dev
```

或者也可以直接下载每天晚上构建出的二进制文件，比如在 64 位系统上，如下所示。

```
$ wget https://experimental.docker.com/builds/Linux/x86_64/docker-latest
$ chmod +x docker-latest
$ ./docker-latest version | grep Version
 Version:      1.8.0-dev
 Version:      1.8.0-dev
```

如果你想默认使用这个二进制文件，请参考范例 4.4 中的说明。

1.10.3　参考

- 最近发布的 Docker 实验版通道（https://blog.docker.com/2015/06/experimental-binary/）
- 运行实验版 Docker 二进制文件（http://docs.docker.com/engine/installation/binaries/）

1.11　在Docker中运行Hello World

1.11.1　问题

你已经拥有一台 Docker 主机，想运行你的第一个容器。你想学习容器的不同生命周期。比如，你想运行一个容器并在其中打印 `hello world`。

1.11.2　解决方案

在命令行提示符下键入 `docker`，将会显示 `docker` 命令的使用方法，如下所示。

```
$ docker
Usage: docker [OPTIONS] COMMAND [arg...]

A self-sufficient runtime for linux containers.

...

Commands:
    attach    Attach to a running container
    build     Build an image from a Dockerfile
    commit    Create a new image from a container's changes
...
    rm        Remove one or more containers
    rmi       Remove one or more images
    run       Run a command in a new container
    save      Save an image to a tar archive
    search    Search for an image on the Docker Hub
    start     Start a stopped container
    stop      Stop a running container
    tag       Tag an image into a repository
    top       Lookup the running processes of a container
    unpause   Unpause a paused container
    version   Show the Docker version information
    wait      Block until a container stops, then print its exit code
```

你已经见过了 `docker ps` 命令，该命令用于列出所有运行中的容器。在本书的其他范例中，你将会看到更多的命令。首先，你想启动一个容器。让我们立即开始，执行 `docker run` 命令，如下所示。

```
$ docker run busybox echo hello world
Unable to find image 'busybox' locally
busybox:latest: The image you are pulling has been verified
511136ea3c5a: Pull complete
df7546f9f060: Pull complete
e433a6c5b276: Pull complete
e72ac664f4f0: Pull complete
Status: Downloaded newer image for busybox:latest
hello world
```

容器来源于镜像。`docker run` 命令也需要一个参数来指定使用哪个镜像。在上面的例子中，使用了名为 busybox 的镜像。在本地 Docker 中还没有这个镜像，所以需要先从公开 registry 中将这个镜像拖到本地。registry 是一个 Docker 镜像的仓库，Docker 客户端可以与这个仓库进行交互并从中下载镜像。镜像下载完毕后，Docker 就会启动容器，并在容器内执行 `echo hello world` 命令。恭喜你成功运行自己的第一个容器。

1.11.3 讨论

如果列出运行中的容器，你会发现没有容器正在运行。这是因为容器一旦完成了它的工作（输出 `hello world`），就停止了。但是容器并没有完全消失，你可以通过 `docker ps -a` 命令看到这个容器，如下所示。

```
$ docker ps
CONTAINER ID    IMAGE    COMMAND      CREATED   STATUS     PORTS        NAMES
$ docker ps -a
CONTAINER ID    IMAGE           COMMAND            ...    PORTS    NAMES
8f7089b187e8    busybox:latest  "echo hello world" ...          thirsty_morse
```

可以看到，输出信息中包括该容器的 ID（8f7089b187e8）和镜像（busybox:latest），还有容器的名称以及容器所运行的命令。你可以通过 `docker rm 8f7089b187e8` 命令将这个容器永久删除。该容器使用的镜像已经被下载到了本地，`docker images` 命令将会输出这个镜像的信息，如下所示。

```
$ docker images
REPOSITORY       TAG             IMAGE ID        CREATED         VIRTUAL SIZE
busybox          latest          e72ac664f4f0    9 weeks ago     2.433 MB
```

如果任何运行中或者已经停止的容器都没有使用这个镜像，你就可以通过 `docker rmi busybox` 命令来删除这个镜像。

运行 echo 命令虽然很有趣，但是获得一个终端会话会更好。要想在容器中运行 `/bin/bash`，你需要使用 `-t` 和 `-i` 参数来获得一个交互式会话，下面以使用 Ubuntu 镜像为例进行说明。

```
$ docker run -t -i ubuntu:14.04 /bin/bash
Unable to find image 'ubuntu:14.04' locally
ubuntu:14.04: The image you are pulling has been verified
01bf15a18638: Pull complete
30541f8f3062: Pull complete
e1cdf371fbde: Pull complete
9bd07e480c5b: Pull complete
511136ea3c5a: Already exists
Status: Downloaded newer image for ubuntu:14.04
root@6f1050d21b41:/#
```

你会看到 Docker 下载的 ubuntu:14.04 镜像由多个层组成，然后你得到了一个容器中 root 权限的会话。提示符也显示了这个容器的 ID。一旦你退出这个终端，该容器就会停止运行，就像我们前面的 hello world 例子一样。

如果你错过了最开始关于如何安装 Docker 的范例，也可以试试网页版的模拟器（https://www.docker.com/tryit/）。这个工具提供了一个 10 分钟的 Docker 教程，你可以通过这个网页初步了解一下 Docker。

1.12　以后台方式运行Docker容器

1.12.1　问题

你已经知道如何以交互方式启动一个容器，但是你想以后台方式运行一个服务。

1.12.2 解决方案

使用 `docker run` 的 `-d` 选项。

运行下面的命令，将会在容器中启动一个简单的基于 Python 的 HTTP 服务器。这个容器使用了 python:2.7 镜像，该镜像下载自 Docker Hub（https://registry.hub.docker.com/_/python/，参见范例 2.9）。

```
$ docker run -d -p 1234:1234 python:2.7 python -m SimpleHTTPServer 1234
$ docker ps
CONTAINER ID  IMAGE       COMMAND               ...   NAMES
0fae2d2e8674  python:2.7  "python -m SimpleHTT   ...   suspicious_pike
```

如果你在浏览器上访问该容器所在主机的 IP 地址以及 1234 端口，将会看到该容器中根目录下的所有文件列表。Docker 会根据 `-p 1234:1234` 参数自动在容器和宿主机之间进行端口映射。在范例 3.2 中，你将会详细了解 Docker 的这一网络功能。

1.12.3 讨论

这个 `-d` 参数会让容器在后台运行。你可以通过运行 `exec` 命令来启动一个 bash shell，再次进入到该容器中，如下所示。

```
$ docker exec -ti 9d7cebd75dcf /bin/bash
root@9d7cebd75dcf:/# ps -ef | grep python
root         1     0  0 15:42 ?        00:00:00 python -m SimpleHTTPServer 1234
```

在官方文档（https://docs.docker.com/reference/run/）中还有 `docker run` 的其他很多选项。你可以实验一下给容器指定名称，修改容器中的工作目录，设置一个环境变量，等等。

1.12.4 参考

- Docker run 参考文档（https://docs.docker.com/reference/run/）

1.13 创建、启动、停止和移除容器

1.13.1 问题

你已经知道如何启动一个容器并让它在后台运行。你希望学习基本命令来管理容器的整个生命周期。

1.13.2 解决方案

使用 Docker 命令行的 `create`、`start`、`stop`、`kill` 和 `rm` 命令。你可以在这些命令后面加上 `-h` 或者 `--h` 选项来查看它们的使用方法，或者只输入命令而不指定任何参数（比如 `docker create`）。

1.13.3 讨论

在范例 1.12 中，你通过 docker run 自动启动了一个容器。你也可以通过 docker create 命令来创建一个容器。继续使用上面简单的 HTTP 服务器的例子，唯一的区别就是这里没有指定 -d 选项。当创建容器之后，你需要运行 docker start 来启动这个容器，如下所示。

```
$ docker create -P --expose=1234 python:2.7 python -m SimpleHTTPServer 1234
a842945e2414132011ae704b0c4a4184acc4016d199dfd4e7181c9b89092de13
$ docker ps -a
CONTAINER ID  IMAGE       COMMAND              CREATED       ... NAMES
a842945e2414  python:2.7  "python -m SimpleHTT 8 seconds ago ... fervent_hodgkin
$ docker start a842945e2414
a842945e2414
$ docker ps
CONTAINER ID  IMAGE       COMMAND              ...   NAMES
a842945e2414  python:2.7  "python -m SimpleHTT ...   fervent_hodgkin
```

要想停止一个正在运行中的容器，可以选择使用 docker kill（这个命令会发送 SIGKILL 信号到容器）或者 docker stop（这个命令会发送 SIGTERM 到容器，如果在一定时间内容器还没有停止，则会再发送 SIGKILL 信号强制停止）。这两个命令最终的结果是停止容器的运行，该容器将不会出现在 docker ps 返回的运行中容器列表中。但是，容器还没有完全消失（比如容器的文件系统还在）。你可以通过 docker restart 来重启这个容器，或者通过 docker rm 移除这个容器，如下所示。

```
$ docker restart a842945e2414
a842945e2414
$ docker ps
CONTAINER ID    IMAGE       COMMAND              ...   NAMES
a842945e2414    python:2.7  "python -m SimpleHTT ...   fervent_hodgkin
$ docker kill a842945e2414
a842945e2414
$ docker rm a842945e2414
a842945e2414
$ docker ps -a
CONTAINER ID    IMAGE       COMMAND     CREATED     STATUS      PORTS       NAMES
```

如果你有很多停止中的容器待删除，可以在一条命令中使用嵌套的 shell 来删除所有容器。docker ps 的 -q 选项只会返回容器的 ID 信息，如下所示。

```
$ docker rm $(docker ps -a -q)
```

1.14 使用Dockerfile构建Docker镜像

1.14.1 问题

你知道了如何从公有的 Docker registry 下载镜像，但是你想构建自己的 Docker 镜像。

1.14.2 解决方案

使用 Dockerfile 构建镜像。Dockerfile 是一个文本文件，它记述了 Docker 构建一个镜像所需要的过程，包括安装软件包、创建文件夹、定义环境变量以及其他一些操作。在第 2 章中我们会对 Dockerfile 和构建镜像做更深入的说明。本范例中只会涉及构建镜像的基本概念。

作为一个简单例子，我们假设你要基于 busybox 镜像创建一个新镜像，并定义一个环境变量。busybox 镜像是一个包含了 busybox（http://www.busybox.net/about.html）二进制文件的 Docker 镜像，这个二进制文件将很多 Unix 实用工具打包到了一个单一的二进制文件中。在一个空文件夹下创建一个名为 Dockerfile 的文件，如下所示。

```
FROM busybox

ENV foo=bar
```

可以通过 `docker build` 命令来构建一个新镜像，并命名为 busybox2，如下所示。

```
$ docker build -t busybox2 .
Sending build context to Docker daemon 2.048 kB
Step 0 : FROM busybox
latest: Pulling from library/busybox
cf2616975b4a: Pull complete
6ce2e90b0bc7: Pull complete
8c2e06607696: Pull complete
Digest: sha256:df9e13f36d2d5b30c16bfbf2a6110c45ebed0bfa1ea42d357651bc6c736d5322
Status: Downloaded newer image for busybox:latest
 ---> 8c2e06607696
Step 1 : ENV foo bar
 ---> Running in f46c59e9bdd6
 ---> 582bacbe7aaa
```

构建结束之后，你就能通过 `docker images` 命令看到新构建的镜像了。可以基于这个新镜像启动一个容器，检查一下其中是否有一个名为 `foo` 的环境变量，并且其值被设置为了 `bar`，如下所示。

```
$ docker images
REPOSITORY      TAG        IMAGE ID        CREATED         VIRTUAL SIZE
busybox2        latest     582bacbe7aaa    6 seconds ago   2.433 MB
busybox         latest     8c2e06607696    3 months ago    2.433 MB
$ docker run busybox2 env | grep foo
foo=bar
```

1.14.3 参考

- Dockerfile 参考指南（https://docs.docker.com/reference/builder/）
- 第 2 章，其中会对创建镜像和共享镜像进行说明

1.15 在单一容器中使用Supervisor运行WordPress

1.15.1 问题

你已经知道了如何将两个容器链接到一起（参见范例 1.16），不过你希望在一个容器中运行应用程序所需的所有服务。以运行 WordPress 为例，你想在一个容器中同时运行 MySQL 和 HTTPD 服务。由于 Docker 运行的是前台进程，所以你需要找到一种同时运行多个“前台”进程的方式。

1.15.2 解决方案

使用 Supervisor（http://supervisord.org/index.html）来监控并运行 MySQL 和 HTTPD。Supervisor 不是一个 init 系统，而是一个用来控制多个进程的普通程序。

> 本范例是一个在容器中使用 Supervisor 同时运行多个进程的例子。你可以以此为基础在一个 Docker 镜像中运行多个服务（比如 SSH、Nginx）。本范例中，WordPress 的配置是一个最精简的可行配置，并不适用于生产环境。

示例中的文件可以在 GitHub（https://github.com/how2dock/docbook/tree/master/ch01/supervisor）下载。这些文件中包括一个用于启动虚拟机的 Vagrantfile，Docker 运行在该虚拟机中，还包含一个 Dockerfile 来定义要创建的镜像，此外还有一个 Supervisor 的配置文件（supervisord.conf）和一个 WordPress 的配置文件（wp-config.php）。

> 如果你不想使用 Vagrant，也可以使用其中的 Dockerfile、`supervisord` 和 WordPress 的配置文件，在自己的 Docker 主机上来安装。

为了运行 WordPress，你需要安装 MySQL、Apache 2（即 `httpd`）、PHP 以及最新版本的 WordPress。你将需要创建一个用于 WordPress 的数据库。在该范例的配置文件中，WordPress 数据库用户名为 root，密码也是 root，数据库名为 wordpress。如果你想对数据库的配置进行修改，需要同时修改 wp-config.php 和 Dockerfile 这两个文件，并让它们的设置保持一致。

Dockerfile 文件用来描述一个 Docker 镜像是如何构建的，后面章节会有关于 Dockerfile 的详细说明。如果这是你第一次使用 Dockerfile 文件，那么你可以直接使用下面的文件，以后再学习 Dockerfile（参见范例 2.3 对 Dockerfile 的介绍）。

```
FROM ubuntu:14.04

RUN apt-get update && apt-get -y install \
    apache2 \
```

```
        php5 \
        php5-mysql \
        supervisor \
        wget

    RUN echo 'mysql-server mysql-server/root_password password root' | \
        debconf-set-selections && \
        echo 'mysql-server mysql-server/root_password_again password root' | \
        debconf-set-selections

    RUN apt-get install -qqy mysql-server

    RUN wget http://wordpress.org/latest.tar.gz && \
        tar xzvf latest.tar.gz && \
        cp -R ./wordpress/* /var/www/html && \
        rm /var/www/html/index.html

    RUN (/usr/bin/mysqld_safe &); sleep 5; mysqladmin -u root -proot create wordpress

    COPY wp-config.php /var/www/html/wp-config.php
    COPY supervisord.conf /etc/supervisor/conf.d/supervisord.conf

    EXPOSE 80

    CMD ["/usr/bin/supervisord"]
```

Supervisor 的配置文件 supervisord.conf 如下所示。

```
[supervisord]
nodaemon=true

[program:mysqld]
command=/usr/bin/mysqld_safe
autostart=true
autorestart=true
user=root

[program:httpd]
command=/bin/bash -c "rm -rf /run/httpd/* && /usr/sbin/apachectl -D FOREGROUND"
```

这里定义了两个被监控和运行的服务：`mysqld` 和 `httpd`。每个程序都可以使用诸如 `autorestart` 和 `autostart` 等选项。最重要的指令是 `command`，其定义了如何运行每个程序。在这个例子中，Docker 容器只需要运行一个前台进程：`supervisord`。从 Dockerfile 中的 `CMD [“/usr/bin/supervisord”]` 这一行也能看出来。

在你的 Docker 主机上，构建该镜像并启动一个后台容器。如果按照例子中的配置文件使用了基于 Vagrant 的虚拟机，可以执行如下命令。

```
$ cd /vagrant
$ docker build -t wordpress .
$ docker run -d -p 80:80 wordpress
```

容器启动后还会在宿主机和 Docker 容器之间为 80 端口进行端口映射。现在只需要在浏览器中打开 http://<ip_of_docker_host>，就可以进入到 WordPress 的配置页面了。

1.15.3 讨论

尽管通过 Supervisor 在一个容器内同时运行多个应用服务工作起来非常完美，但是你最好还是使用多个容器来运行不同的服务。这能充分利用容器的隔离优势，也能帮助你创建基于微服务设计思想的应用（参见《微服务设计》[1]）。最终这也将会使你的应用更具弹性和可扩展性。

1.15.4 参考

- Supervisor 文档（http://supervisord.org/index.html）
- Docker Supervisor 文档（https://docs.docker.com/articles/using_supervisord/）

1.16 使用两个链接在一起的容器运行WordPress博客程序

1.16.1 问题

你希望通过容器来运行一个 WordPress 网站（http://wordpress.com/），但是你不想让 MySQL 和 WordPress 在同一个容器中运行。你时刻谨记对关注点进行分离的原则，并尽可能地对应用程序的不同组件进行解耦。

1.16.2 解决方案

启动两个容器：一个运行来自 Docker Hub（http://hub.docker.com/）的官方 WordPress，另一个运行 MySQL 数据库。这两个容器通过 Docker 命令行工具的 `--link` 选项链接在一起。

开始下载最新的 WordPress（https://hub.docker.com/_/wordpress/）和 MySQL（https://.hub.docker.com/_/mysql/）镜像，如下所示。

```
$ docker pull wordpress:latest
$ docker pull mysql:latest
$ docker images
REPOSITORY      TAG           IMAGE ID       CREATED        VIRTUAL SIZE
mysql           latest        9def920de0a2   4 days ago     282.9 MB
wordpress       latest        93acfaf85c71   8 days ago     472.8 MB
```

启动一个 MySQL 容器，并通过命令行工具的 `--name` 选项为这个容器设置一个名称，通过 `MYSQL_ROOT_PASSWORD` 环境变量来设置 MySQL 的密码，如下所示。

```
$ docker run --name mysqlwp -e MYSQL_ROOT_PASSWORD=wordpressdocker -d mysql
```

注 1：此书已由人民邮电出版社出版。——编者注

> 如果在使用 mysql 镜像时没有指定标签，Docker 会自动使用 `latest` 标签，这也是前面刚刚下载的镜像。容器通过 `-d` 选项以守护式的方式开始运行。

现在就可以基于 wordpress:latest 镜像启动 WordPress 容器了。这个容器将会通过 `--link` 选项链接到 MySQL 容器，这样 Docker 会自动进行网络配置，让 WordPress 容器能访问到 MySQL 容器中暴露出来的端口，如下所示。

```
$ docker run --name wordpress --link mysqlwp:mysql -p 80:80 -d wordpress
```

两个容器都会以后台方式运行，WordPress 容器的 80 端口会被映射到宿主机的 80 端口上，如下所示。

```
$ docker ps
CONTAINER ID    IMAGE             COMMAND              CREATED             ...
e1593e7a20df    wordpress:latest  "/entrypoint.sh apac  About a minute ago  ...
d4be18e33153    mysql:latest      "/entrypoint.sh mysq  5 minutes ago       ...

...             STATUS             PORTS                NAMES
...             Up About a minute  0.0.0.0:80->80/tcp   wordpress
...             Up 5 minutes       3306/tcp             mysqlwp
```

在浏览器中打开 http://<ip_of_host> 就会看到 WordPress 的安装界面，里面有选择语言的窗口，如图 1-10 所示。完成了 WordPress 的安装过程，你将会得到一个在两个链接到一起的容器之上运行的 WordPress 网站。

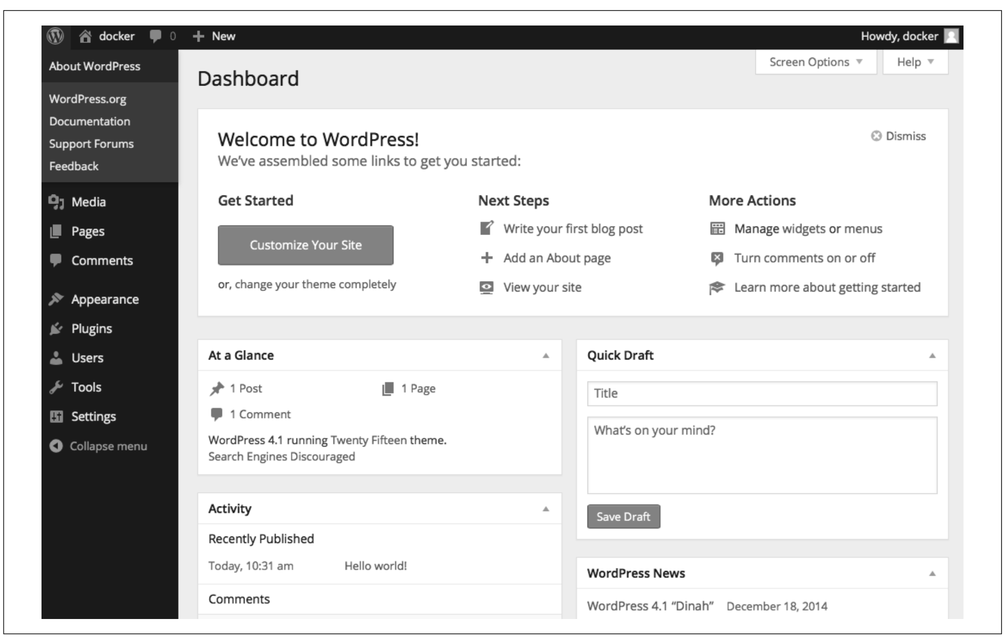


图 1-10：在容器中运行的 WordPress 站点

1.16.3　讨论

这两个 WordPress 和 MySQL 镜像都是官方镜像，分别由 WordPress 和 MySQL 的社区来维护。Docker Hub 这些镜像的页面都有关于如何进行配置以从这些镜像创建容器的详细文档。

> 别忘了阅读 WordPress 镜像文档（https://hub.docker.com/_/wordpress/）和 MySQL 镜像文档（https://hub.docker.com/_/mysql/）。

令人感兴趣的是，你可以通过设置几个环境变量来创建一个数据库，并且只有具有相应权限的用户才能操作数据库：`MYSQL_DATABASE`、`MYSQL_USER` 和 `MYSQL_PASSWORD`。在前面的例子中，WordPress 使用了 MySQL 的 root 用户，这并不是一个好实践。最好是创建一个名为 `wordpress` 的数据库，并为其创建一个用户，像下面这样。

```
$ docker run --name mysqlwp -e MYSQL_ROOT_PASSWORD=wordpressdocker \
                            -e MYSQL_DATABASE=wordpress \
                            -e MYSQL_USER=wordpress \
                            -e MYSQL_PASSWORD=wordpresspwd \
                            -d mysql
```

> 如果你需要删除所有容器，可以使用下面这种嵌套 shell 的快捷方式。
>
> ```
> $ docker stop $(docker ps -q)
> $ docker rm -v $(docker ps -aq)
> ```
>
> `docker rm` 命令的 `-v` 选项用来删除 MySQL 镜像中定义的数据卷。

数据库容器启动之后，可以启动 WordPress 容器并指定你设置好的数据库表，如下所示。

```
$ docker run --name wordpress --link mysqlwp:mysql -p 80:80 \
                              -e WORDPRESS_DB_NAME=wordpress \
                              -e WORDPRESS_DB_USER=wordpress \
                              -e WORDPRESS_DB_PASSWORD=wordpresspwd \
                              -d wordpress
```

1.17　备份在容器中运行的数据库

1.17.1　问题

你使用 MySQL 镜像对外提供数据库服务。为了对数据进行持久化，你需要备份该数据库。

1.17.2　解决方案

你可以使用一种或几种备份策略结合起来使用。适用于容器的主要有两种方式：一是可以

在一个以后台方式运行的容器中执行一条命令，二是挂载一个宿主机卷（即一个在宿主机上能访问的存储区域）到容器中。在本范例中，我们将会看到如何去做下面两件事：

- 将 Docker 主机上的卷挂载到 MySQL 容器中
- 使用 `docker exec` 命令执行 `mysqldump`

在范例 1.16 的例子中，我们通过两个链接在一起的容器安装了一个 WordPress 站点。在该范例中，我们将会修改启动 MySQL 容器的方式。容器启动之后，一个功能齐全的 WordPress 网站已经安装完成，你可以停止容器，这也将会停止你的应用程序。这时容器还没有被完全删除，数据库中的数据仍然可以访问到。不过，如果你删除容器（`docker rm $(docker ps -aq)`），那么其中所有的数据都会丢失。

有一种方式可以在容器被 `docker rm -v` 命令删除之后也能保留数据，就是将宿主机的卷挂载到容器中。如果你只是使用 `docker rm` 命令来删除容器，那么这个容器的镜像所定义的卷会在磁盘上保留，尽管这时容器已经不存在了。如果看一下构建这个 MySQL 镜像的 Dockerfile 文件（https://github.com/docker-library/mysql/blob/d6268ace61047c74468d7c59b4d8da6be5dec16a/5.6/Dockerfile），将会看到 `VOLUME /var/lib/mysql` 这一行。这一行的意思是，当你基于该镜像启动一个容器时，可以将宿主机的文件夹绑定到容器中的这个挂载点上，比如下面这样。

```
$ docker run --name mysqlwp -e MYSQL_ROOT_PASSWORD=wordpressdocker \
                            -e MYSQL_DATABASE=wordpress \
                            -e MYSQL_USER=wordpress \
                            -e MYSQL_PASSWORD=wordpresspwd \
                            -v /home/docker/mysql:/var/lib/mysql \
                            -d mysql
```

上面命令中 `-v /home/docker/mysql:/var/lib/mysql` 这一行进行了宿主机和容器中卷的绑定。当完成 WordPress 的设置之后，在宿主机 /home/docker/mysql 文件夹下就能看到这些文件变动。

```
$ ls mysql/
auto.cnf  ibdata1  ib_logfile0  ib_logfile1  mysql  performance_schema  wordpress
```

为了对整个 MySQL 数据库进行备份，可以使用 `docker exec` 命令在容器内执行 `mysqldump`，如下所示。

```
$ docker exec mysqlwp mysqldump --all-databases \
                                --password=wordpressdocker > wordpress.backup
```

现在你可以使用传统方式来进行数据库的备份和恢复了。比如，在云环境中，你可能会将 Elastic Block Store（例如 AWS EBS）挂载到一个主机实例，再挂载到容器中。你也可以将你的 MySQL 备份保存到 Elastic Storage（比如 AWS S3）中。

1.17.3 讨论

尽管本例中使用的是 MySQL 数据库，不过这种方式也适用于 Postgres 和其他数据库。如果你用了 Docker Hub 上的 Postgres（https://registry.hub.docker.com/_/postgres/）镜像，那么

也可以从它的 Dockerfile（https://github.com/docker-library/postgres/blob/5f401753715311752f2346cdbd32bd55e7251ddd/9.4/Dockerfile）中看到其中也定义了一个卷（`VOLUME /var/lib/postgresql/data`）。

1.18 在宿主机和容器之间共享数据

1.18.1 问题

你在宿主机上有一些数据，想在容器中也能访问到这些数据。

1.18.2 解决方案

在运行 `docker run` 命令时，通过设置 `-v` 选项将宿主机的卷挂载到容器中。

比如，你想将宿主机 /cookbook 目录下的工作目录与容器共享，可以执行以下指令。

```
$ ls
data
$ docker run -ti -v "$PWD":/cookbook ubuntu:14.04 /bin/bash
root@11769701f6f7:/# ls /cookbook
data
```

在这个例子中，宿主机上的当前工作目录会挂载到容器中的 /cookbook 目录上。如果你在容器内创建了文件或者文件夹，那么这些修改会直接反映到宿主机上，如下所示。

```
$ docker run -ti -v "$PWD":/cookbook ubuntu:14.04 /bin/bash
root@44d71a605b5b:/# touch /cookbook/foobar
root@44d71a605b5b:/# exit
exit
$ ls -l foobar
-rw-r--r-- 1 root root 0 Mar 11 11:42 foobar
```

默认情况下，Docker 会以读写模式挂载数据卷。如果想以只读方式挂载数据卷，可以在卷名称后通过冒号设置相应的权限。比如在前面的例子中，如果想以只读方式将工作目录挂载到 /cookbook，可以使用 `-v "$PWD":/cookbook:ro`。可以通过 `docker inspect` 命令来查看数据卷的挂载映射情况。参考范例 9.1 可以获取更多有关 `inspect` 的介绍。

```
$ docker inspect -f {{.Mounts}} 44d71a605b5b
[{ /Users/sebastiengoasguen/Desktop /cookbook   true}]
```

1.18.3 参考

- 管理容器内的数据（https://docs.docker.com/userguide/dockervolumes/）
- 了解卷（http://container-solutions.com/2014/12/understanding-volumes-docker/）
- 数据容器（http://container42.com/2014/11/18/data-only-container-madness/）
- Docker volume（http://container42.com/2014/11/03/docker-indepth-volumes/）

1.19 在容器之间共享数据

1.19.1 问题

你已经知道了如何将宿主机中的数据卷挂载到运行中的容器上，但是你想和其他容器共享在一个容器中定义的卷。这样做的优点是让 Docker 去负责管理卷，并且遵循了单一职责这一原则。

1.19.2 解决方案

使用数据容器。在范例 1.18 中，你已经知道了如何将宿主机中的卷挂载到容器中，可以使用 `docker run` 命令的 `-v` 选项指定宿主机中的卷，以及该卷在容器中要挂载的路径。如果省略了宿主机中的路径，那么你就创建了一个称为*数据容器*的容器。

容器启动后会在内部创建参数中指定的卷，这是一个具备读写权限的文件系统，它不在容器镜像最顶层的只读层之上。Docker 会负责管理这个文件系统，但你也可以在宿主机上对其进行读写。下面我们就来看看它是如何工作的。（为了方便说明，这里对卷的 ID 进行了截断。）

```
$ docker run -ti -v /cookbook ubuntu:14.04 /bin/bash
root@b5835d2b951e:/# touch /cookbook/foobar
root@b5835d2b951e:/# ls cookbook/
foobar
root@b5835d2b951e:/# exit
exit
bash-4.3$ docker inspect -f {{.Mounts}} b5835d2b951e
[{dbba7caf8d07b862b61b39... /var/lib/docker/volumes/dbba7caf8d07b862b61b39... \
/_data /cookbook local  true}]
$ sudo ls /var/lib/docker/volumes/dbba7caf8d07b862b61b39...
foobar
```

> 由 Docker 引擎创建的用于存储卷数据的目录位于 Docker 主机之上。如果你是通过 Docker Machine 创建的远程 Docker 主机，就需要连接到该远程 Docker 主机来查看 Docker volume 数据路径的详情。

这个容器启动之后，Docker 会创建 /cookbook 目录。在容器中，你可以对这个目录进行读写操作。在你退出这个容器之后，可以通过 `inspect` 命令（参见范例 9.1）来查看这个数据卷被保存到了宿主机的什么位置。Docker 会在 /var/lib/docker/volumes/ 下为这个卷创建对应的文件夹。你可以在宿主机上对这个文件夹进行读写。任何对这个文件夹的修改都会被保存下来，如果你重启了这个容器，那么在容器内也能看到修改后的内容，如下所示。

```
$ sudo touch /var/lib/docker/volumes/dbba7caf8d07b862b61b39.../foobar2
$ docker start b5835d2b951e
$ docker exec -ti b5835d2b951e /bin/bash
root@b5835d2b951e:/# ls /cookbook
foobar  foobar2
```

为了将这个容器中的卷共享给其他容器，可以使用 --volumes-from 选项。让我们重新开始来创建一个数据容器，然后再创建另一个容器来挂载源数据容器中共享的卷，如下所示。

```
$ docker run -v /data --name data ubuntu:14.04
$ docker ps
CONTAINER ID    IMAGE   COMMAND     CREATED   STATUS   PORTS    NAMES
$ docker inspect -f {{.Mounts}} data
[{4ee1d9e3d453e843819c6ff... /var/lib/docker/volumes/4ee1d9e3d453e843819c6ff... \
/_data /data local true]
```

> 这个容器并没有处于运行状态。但是它的卷映射关系已经存在，并且卷被持久化到了 /var/lib/docker/vfs/dir 下面。你可以通过 `docker rm -v data` 命令来删除容器和它的卷。如果你没有使用 `rm -v` 选项来删除容器和它的卷，那么系统中将会遗留很多没有被使用的卷。

即使这个数据容器没有运行，你也可以通过 --volumes-from 来挂载其中的卷，如下所示。

```
$ docker run -ti --volumes-from data ubuntu:14.04 /bin/bash
root@b94a006377c1:/# touch /data/foobar
root@b94a006377c1:/# exit
exit
$ sudo ls /var/lib/docker/volumes/4ee1d9e3d453e843819c6ff...
foobar
```

1.19.3 参考

- 了解卷（http://container-solutions.com/2014/12/understanding-volumes-docker/）
- 数据容器（http://container42.com/2014/11/18/data-only-container-madness/）
- Docker volume（http://container42.com/2014/11/03/docker-indepth-volumes/）
- Docker 官方文档（https://docs.docker.com/userguide/dockervolumes/）
- Docker volume 的优点（https://medium.com/@ramangupta/why-docker-data-containers-are-good-589b3c6c749e#.fabqztipw）

1.20 对容器进行数据复制

1.20.1 问题

你有一个运行中的容器，没有设置任何卷映射信息，但是你想从容器内复制数据出来或者将数据复制到容器里。

1.20.2 解决方案

使用 `docker cp` 命令将文件从正在运行的容器复制到 Docker 主机。`docker cp` 命令支持在 Docker 主机与容器之间进行文件复制。其用法很简单，如下所示。

```
$ docker cp
docker: "cp" requires 2 arguments.
```

```
See 'docker cp --help'.

Usage:  docker cp [OPTIONS] CONTAINER:PATH LOCALPATH|-
  docker cp [OPTIONS] LOCALPATH|- CONTAINER:PATH

Copy files/folders between a container and your host.
...
```

为了讲解它是如何工作的，我们先启动一个容器，并让它执行睡眠操作。然后你可以进入到这个容器，并手动创建一个文件，如下所示。

```
$ docker run -d --name testcopy ubuntu:14.04 sleep 360
$ docker exec -ti testcopy /bin/bash
root@b81793e9eb3e:/# cd /root
root@b81793e9eb3e:~# echo 'I am in the container' > file.txt
root@b81793e9eb3e:~# exit
```

要将在容器中创建的这个文件复制到宿主机上，使用 `docker cp` 即可，如下所示。

```
$ docker cp testcopy:/root/file.txt .
$ cat file.txt
I am in the container
```

要想将文件从宿主机复制到容器，仍然可以使用 `docker cp` 命令，只不过源和目的文件的参数要调换一下位置，如下所示。

```
$ echo 'I am in the host' > host.txt
$ docker cp host.txt testcopy:/root/host.txt
```

一个比较好的使用场景是将一个容器中的文件复制到另一个容器中，这可以通过临时先将主机作为文件的中转站，执行两次 `docker cp` 命令来实现。比如，如果想将 /root/file.txt 从两个运行容器中的 c1 复制到 c2，可以使用下面的命令。

```
$ docker cp c1:/root/file.txt .
$ docker file.txt c2:/root/file.txt
```

1.20.3　讨论

如果是 1.8 版本之前的 Docker，`docker cp` 还不支持将文件从宿主机复制到容器中。不过你可以组合使用 `docker exec` 和一些 shell 重定向，如下所示。

```
$ echo 'I am in the host' > host.txt
$ docker exec -i testcopy sh -c 'cat > /root/host.txt' < host.txt
$ docker exec -i testcopy sh -c 'cat /root/host.txt'
I am in the host
```

现在，新版本的 Docker 已经不需要再像上面那样麻烦了，但是上面的例子很好地展示了 `docker exec` 命令的强大之处。

1.20.4　参考

- 该范例的灵感来源于 Grigoriy Chudnov（https://medium.com/@gchudnov/copying-data-between-docker-containers-26890935da3f#.90yqi6xvl）

第 2 章

创建和共享镜像

2.0 简介

在了解了 Docker 的基本用法之后，你可能立刻想要创建自己的镜像。也许你会打包现有的应用程序，或者想从头开始利用 Docker 构建一个全新的镜像。本章将会介绍如何构建 Docker 镜像，以及如何跟别人共享你的镜像。

创建镜像的第一种方法是使用 `docker commit` 命令。你可以使用基础镜像启动一个容器，并以交互方式对容器进行更改。Docker 可以让你通过提交（commit）将这些变更保存到一个新的镜像中（参见范例 2.1）。在这些操作的背后，Docker 会使用联合文件系统创建一个镜像层，来保存基础镜像和新创建镜像之间的差异。你可以简单地将这个镜像导出为 tar 文件后发送给他人来共享这个镜像（参见范例 2.2）。

但是通过手动修改容器然后提交它们来创建镜像并不适合再现，而且也不是自动化的。一个更好的方法是创建一个 Dockerfile 文件，然后让 Docker 自动构建镜像（参考范例 2.3）。你可以在范例 2.4 中看到如何为基于 Python 的简单的 Flask 应用程序创建 Dockerfile，在范例 2.5 中，你将会学习到关于优化 Dockerfile 的最佳实践。

如果你之前曾经使用过 Vagrant 和 Packer，那么你可能会想要浏览一下范例 2.7 和范例 2.8。这样，你就可以重用你已有的配置管理资产构建 Docker 镜像，且更容易接受 Docker。

通过导出和导入功能共享镜像虽然也能很好地工作，但是你应该利用 Docker Hub 来与他人共享镜像，将 Docker 集成到持续集成工作流中。Docker Hub（参见范例 2.9）是一个应用程序镜像仓库。Docker Hub 上的镜像可以公开共享给他人，也可以通过与代码托管服务（比如 GitHub 和 Bitbucket）集成进行自动镜像构建（参见范例 2.12）。

最后，如果你不想使用 Docker Hub，也可以部署自己私有的 Docker 镜像 registry（参见范

例 2.11），并建立自己的自动化构建（范例 2.13）。

学完本章，你将能够为不同的应用程序和服务编写 Dockerfile，并通过类似 Docker Hub 这样的托管服务或者你自己的 Docker registry 共享镜像。这将帮助你快速地搭建持续集成和部署工作流。

2.1 将对容器的修改提交到镜像

2.1.1 问题

在容器内部进行一些修改之后，你想将这些修改保存下来。你不想在退出或者停止这个容器后丢失这些修改，并且你还想将这些修改作为其他容器的基础进行重用。

2.1.2 解决方案

通过 `docker commit` 命令提交你对容器作出的修改，并创建一个新镜像。

让我们以交互式 bash shell 的方式启动一个容器，并更新其中的软件包，如下所示。

```
$ docker run -t -i ubuntu:14.04 /bin/bash
root@69079aaaaab1:/# apt-get update
```

当你退出容器后，容器会停止运行，但容器还在，直到你通过 `docker rm` 命令彻底将容器从系统中删除。所以在删除容器之前，可以提交对容器作出的修改，并以此创建一个新的镜像 ubuntu:update。镜像的名称为 ubuntu，同时添加了一个标签 `update`（参见范例 2.6），以与 ubuntu:latest 镜像加以区分。

```
$ docker commit 69079aaaaab1 ubuntu:update
13132d42da3cc40e8d8b4601a7e2f4dbf198e9d72e37e19ee1986c280ffcb97c
$ docker images
REPOSITORY          TAG               IMAGE ID         CREATED          VIRTUAL SIZE
ubuntu              update            13132d42da3c     5 days ago       213 MB
...
```

现在你就可以安全地删除已经停止的容器了，同时也可以基于刚创建的 ubuntu:update 镜像启动新的容器。

2.1.3 讨论

可以通过 `docker diff` 命令来查看在容器中对镜像作出的修改，如下所示。

```
$ docker diff 69079aaaaab1
C /root
A /root/.bash_history
C /tmp
C /var
C /var/cache
C /var/cache/apt
D /var/cache/apt/pkgcache.bin
```

```
D /var/cache/apt/srcpkgcache.bin
C /var/lib
C /var/lib/apt
C /var/lib/apt/lists
...
```

`A` 表示文件或者文件夹是新增加的，`C` 表示文件内容有修改，`D` 则表示该项目已经删除。

2.1.4 参考

- `docker commit` 参考手册（https://docs.docker.com/reference/commandline/cli/#commit）
- `docker diff` 参考手册（https://docs.docker.com/reference/commandline/cli/#diff）

2.2 将镜像和容器保存为tar文件进行共享

2.2.1 问题

你创建了一些镜像，或者有一些容器，你希望能将它们保存下来并与你的同伴共享。

2.2.2 解决方案

对于已有镜像，可以使用 Docker 命令行的 `save` 和 `load` 命令来创建一个压缩包（tarball）；而对于容器，可以使用 `import` 和 `export` 进行导入导出操作。

让我们从一个停止的容器开始，将它导出为一个新的压缩包文件，如下所示。

```
$ docker ps -a
CONTAINER ID    IMAGE            COMMAND     CREATED          ...    NAMES
77d9619a7a71    ubuntu:14.04    "/bin/bash"  10 seconds ago   ...    high_shockley
$ docker export 77d9619a7a71 > update.tar
$ ls
update.tar
```

可以在本地将容器提交为一个新镜像（参见范例 2.1），但是也可以使用 Docker `import` 命令，如下所示。

```
$ docker import - update < update.tar
157bcbb5fdfce0e7c10ef67ebdba737a491214708a5f266a3c74aa6b0cfde078
$ docker images
REPOSITORY          TAG                 IMAGE ID       ...    VIRTUAL SIZE
update              latest              157bcbb5fdfc   ...    188.1 MB
```

如果想和你的同伴共享这个镜像，可以将这个镜像上传到一个 Web 服务器，你的同伴将这个镜像下载之后再通过 Docker 的 `import` 命令将镜像导入本地即可。

如果要对你通过提交操作创建的镜像进行导入导出，可以使用 `load` 和 `save` 命令，如下所示。

```
$ docker save -o update1.tar update
$ ls -l
total 385168
```

```
-rw-rw-r-- 1 vagrant vagrant 197206528 Jan 13 14:13 update1.tar
-rw-rw-r-- 1 vagrant vagrant 197200896 Jan 13 14:05 update.tar
$ docker rmi update
Untagged: update:latest
Deleted: 157bcbb5fdfce0e7c10ef67ebdba737a491214708a5f266a3c74aa6b0cfde078
$ docker load < update1.tar
$ docker images
REPOSITORY      TAG       IMAGE ID          CREATED           VIRTUAL SIZE
update          latest    157bcbb5fdfc      5 minutes ago     188.1 MB
ubuntu          14.04     8eaa4ff06b53      12 days ago       192.7 MB
```

2.2.3 讨论

这两种方法非常相似。不同的是，保存一个镜像会保留它的历史，而导出容器将对它的历史进行压缩。

2.3 编写你的第一个Dockerfile

2.3.1 问题

以交互方式启动一个容器，在里面对容器进行修改，将修改后的容器提交为镜像，这样也能正常工作（参见范例 2.1）。但是你想自动构建镜像，并且将构建步骤与他人共享。

2.3.2 解决方案

要想自动构建 Docker 镜像，你需要通过一个名为 Dockerfile 的说明文件来描述镜像构建的步骤。这个文本文件使用一组指令来描述以下各项内容：新镜像的基础镜像，为了安装不同的依赖和应用程序需要执行哪些操作步骤，镜像中需要提供哪些文件，这些文件是怎么复制到镜像中的，要暴露哪些端口，以及在新的容器中启动时默认运行什么命令，此外还有一些其他的内容。

为了对此进行说明，让我们开始编写我们的第一个 Dockerfile。从该 Dockerfile 构建的镜像启动新的容器时，会执行 `/bin/echo` 命令。在当前工作目录中创建一个名为 Dockerfile 的文本文件，文件内容如下所示。

```
FROM ubuntu:14.04

ENTRYPOINT ["/bin/echo"]
```

`FROM` 指令指定了新的镜像以哪个镜像为基础开始构建。这里我们选择了 ubuntu:14.04 镜像作为基础镜像。ubuntu:14.04 是来自 Docker Hub 上由 Ubuntu 官方提供的镜像仓库（https://registry.hub.docker.com/_/ubuntu/）。`ENTRYPOINT` 指令设置了从该镜像创建的容器启动时需要执行的命令。要想构建这个镜像，可以在命令行提示符下键入 `docker build .` 命令，如下所示。

```
$ docker build .
Sending build context to Docker daemon  2.56 kB
Sending build context to Docker daemon
Step 0 : FROM ubuntu:14.04
 ---> 9bd07e480c5b
Step 1 : ENTRYPOINT /bin/echo
 ---> Running in da3fa01c973a
 ---> e778362ca7cf
Removing intermediate container da3fa01c973a
Successfully built e778362ca7cf
$ docker images
REPOSITORY          TAG         IMAGE ID       ...    VIRTUAL SIZE
<none>              <none>      e778362ca7cf   ...    192.7 MB
ubuntu              14.04       9bd07e480c5b   ...    192.7 MB
```

现在就可以基于新构建的镜像启动容器了，你需要指定刚刚创建的镜像的 ID 并指定一个参数（即 `Hi Docker !`），如下所示。

```
$ docker run e778362ca7cf Hi Docker !
Hi Docker !
```

非常神奇，你在新容器中执行了 `echo` 命令！这里你基于上面由只有两行的 Dockerfile 构建的镜像创建了一个容器，该容器开始运行并执行了由 `ENTRYPOINT` 指令所定义的命令。当这个命令结束之后，容器的工作也即告结束并退出。如果再次运行上述命令但是不指定任何参数，那么就不会有任何内容回显出来，如下所示。

```
$ docker run e778362ca7cf
```

你也可以在 Dockerfile 文件中使用 `CMD` 指令。使用该指令的优点是，你可以在启动容器时，通过在 `docker run` 命令后面指定新的 `CMD` 参数来覆盖 Dockerfile 文件中设置的内容。让我们使用 `CMD` 指令来构建一个新镜像，如下所示。

```
FROM ubuntu:14.04

CMD ["/bin/echo" , "Hi Docker !"]
```

构建这个镜像并运行它，如下所示。

```
$ docker build .
...
$ docker run eff764828551
Hi Docker !
```

在上面的构建命令中，我们指定了当前文件夹路径。这时候 Docker 会自动使用刚才创建的 Dockerfile 文件。如果希望在构建镜像的时候使用在其他位置保存的 Dockerfile，可以使用 `docker build` 命令的 `-f` 参数来指定 Dockerfile 文件的位置。

上面的操作看起来与之前的例子一模一样，但是，如果在 `docker run` 命令后面指定一个其他的可执行命令，那么该命令就会被执行，而不是执行在 Dockerfile 文件中定义的 `/bin/echo` 命令，如下所示。

```
$ docker run eff764828551 /bin/date
Thu Dec 11 02:49:06 UTC 2014
```

2.3.3 讨论

Dockerfile 是一个文本文件，它定义了一个镜像是如何构建的，以及基于该镜像创建的容器运行时会进行什么处理。通过 FROM、ENTRYPOINT 和 CMD 这三个简单的指令，你已经可以构建一个能完全正常工作的镜像了。当然，我们在该范例中也只是介绍了这三个指令而已，你可以通过阅读 Dockerfile 参考手册（https://docs.docker.com/reference/builder/）学习一下其他指令，或者在范例 2.4 中看一个更详细的例子。

CentOS 项目维护着很多 Dockerfile 的例子。可以在该项目的代码仓库（https://github.com/CentOS/CentOS-Dockerfiles）中查看这些例子，并通过运行其中的一些例子来加深对 Dockerfile 文件的理解。

请记住，CMD 可以通过 docker run 后面的参数来覆盖，而 ENTRYPOINT 只能通过 docker run 的 --entrypoint 参数来覆盖。同时你也看到了，当命令结束时，容器也退出了。也就是说，你希望在容器中运行的进程需要以前台方式来运行；否则，容器会停止。

当第一次构建结束时，新的镜像也构建完成了。在这个例子中，新的镜像 ID 是 e778362ca7cf。请注意，新的镜像并没有仓库名和标签信息，因为在构建时我们并没有指定这些信息。可以重新构建该镜像，通过 docker build 命令的 -t 参数将仓库名设置为 cookbook，将标签设置为 hello。由于这些操作都是在本地进行的，所以你可以随意对镜像仓库和标签进行命名。但是，如果你想将该镜像发布到镜像 registry，就需要遵循一定的命名约定。

```
$ docker build -t cookbook:hello .
$ docker images
REPOSITORY          TAG         IMAGE ID        CREATED         VIRTUAL SIZE
cookbook            hello       e778362ca7cf    4 days ago      192.7 MB
ubuntu              14.04       9bd07e480c5b    10 days ago     192.7 MB
```

docker build 命令有几个选项可以用来设置如何处理构建过程中的临时容器，如下所示。

```
$ docker build -h

Usage: docker build [OPTIONS] PATH | URL | -

Build a new image from the source code at PATH

  --force-rm=false     Always remove intermediate containers...
  --no-cache=false     Do not use cache when building the ...
  -q, --quiet=false    Suppress the verbose output generated...
  --rm=true            Remove intermediate containers after ...
  -t, --tag=""         Repository name (and optionally a tag)...
```

2.3.4　参考

- Dockerfile 参考文档（https://docs.docker.com/reference/builder/）
- 编写 Dockerfile 的最佳实践（https://docs.docker.com/articles/dockerfile_best-practices/）
- CentOS 项目中提供的大量 Dockerfile 文件示例（https://github.com/CentOS/CentOS-Dockerfiles）

2.4　将Flask应用打包到镜像

2.4.1　问题

你有一个基于 Python 框架 Flask（http://flask.pocoo.org）编写的 Web 应用程序，该程序运行于 Ubuntu 14.04 之上。你希望在容器中运行这个应用。

2.4.2　解决方案

这里我们使用一个很简单的 Hello World（http://flask.pocoo.org）应用为例进行说明，其 Python 代码如下所示。

```
#!/usr/bin/env python

from flask import Flask
app = Flask(__name__)

@app.route('/hi')
def hello_world():
    return 'Hello World!'

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000)
```

为了让这个应用程序能够在 Docker 容器中运行，你需要编写一个 Dockerfile 文件，在 Dockerfile 文件中使用 `RUN` 指令来安装运行该程序所需要的软件，使用 `EXPOSE` 指令将应用程序监听的端口暴露给外部。同时你需要使用 `ADD` 指令将应用程序复制到容器内的文件系统上。

这个应用程序的 Dockerfile 如下所示。

```
FROM ubuntu:14.04

RUN apt-get update
RUN apt-get install -y python
RUN apt-get install -y python-pip
RUN apt-get clean all

RUN pip install flask

ADD hello.py /tmp/hello.py
```

```
EXPOSE 5000

CMD ["python","/tmp/hello.py"]
```

这里我们并没有刻意对 Dockerfile 文件进行优化。当你理解了 Dockerfile 的基本概念之后，可以参考范例 2.5 来根据编写 Dockerfile 的最佳实践构建镜像。RUN 指令允许你在构建镜像过程中执行指定的 shell 命令。在这个例子中我们更新了仓库缓存，安装了 Python 和 Pip，然后安装了 Flask 微框架。

通过 ADD 命令可以将应用程序复制到镜像内。这里将 hello.py 文件复制到了 /tmp/ 文件夹下。

这个应用程序使用了 5000 端口，并将这个端口暴露给了 Docker 主机。

最后通过 CMD 指令设置了容器启动时将会执行的命令 python /tmp/hello.py。

剩下的工作就是构建镜像了，如下所示。

```
$ docker build -t flask .
```

该命令将会创建一个 flask Docker 镜像，如下所示。

```
$ docker images
REPOSITORY          TAG         IMAGE ID            CREATED         VIRTUAL SIZE
flask               latest      d381310506ed        4 days ago      354.6 MB
cookbook            echo        e778362ca7cf        4 days ago      192.7 MB
ubuntu              14.04       9bd07e480c5b        10 days ago     192.7 MB
```

启动容器时，使用 docker run 的 -d 选项，这将会让容器以后台守护方式运行，docker run 的 -P 选项告诉 Docker 在宿主机上选择一个端口并映射到在 Dockerfile 中设置的端口（比如 5000）。

```
$ docker run -d -P flask
5ac72ed12a72f0e2bec0001b3e78f11660905d20f40e670d42aee292263cb890
$ docker ps
CONTAINER ID    IMAGE         COMMAND                ... PORTS
5ac72ed12a72    flask:latest  "python /tmp/hello.p  ... 0.0.0.0:49153->5000/tcp
```

该容器会进入后台方式并立即返回，你不会登录到交互式 shell 中。PORTS 字段显示了容器中的 5000 端口被映射到了 Docker 宿主机的 49153 端口上。简单地通过 curl 命令访问 http://localhost:49153/hi 会看到 Hello World 输出结果，或者你也可以在浏览器中打开这个 URL。

> 如果你正在使用 Boot2Docker，那么你需要将 localhost 替换为网桥设备的 IP 地址。如果实在想使用 localhost，可以在 VirtualBox 中增加一条端口转发规则。

2.4.3 讨论

由于你已经在 Dockerfile 中通过 CMD 指令设置了容器要运行的命令，所以在启动容器的时候没有必要在镜像名后面指定要运行的命令。但是你可以覆盖这个默认运行的命令，在启动容器时运行 bash shell 进入交互模式，如下所示。

```
$ docker run -t -i -P flask /bin/bash
root@fc1514ced93e:/# ls -l /tmp
total 4
-rw-r--r-- 1 root root 194 Dec  8 13:41 hello.py
root@fc1514ced93e:/#
```

2.5 根据最佳实践优化Dockerfile

2.5.1 问题

你希望根据最佳实践来编写 Dockerfile，优化你的 Docker 镜像。

2.5.2 解决方案

Docker 文档发布了关于如何编写 Dockerfile 的最佳实践（https://docs.docker.com/articles/dockerfile_best-practices/）。本范例将会选择其中几条来帮助你构建良好的镜像。

(1) 在每个容器中只运行一个进程。虽然你也可以在一个容器中运行多个进程（比如范例 1.15），但是在一个容器中运行一个进程或者单一功能的服务，可以帮助你更好地对应用进行解耦和扩展。利用容器链接（参见范例 3.3）或者其他容器网络技术（参见第 3 章）在多个容器之间进行通信。

(2) 不要以为你的容器将会长久存在：它们是临时的，会被停止和重新启动。你应该把它们当作不可变的实体，这意味着你不应该对其进行修改，而应从基础镜像重新创建它们。因此，需要将运行时配置和数据独立于容器和镜像进行管理。你可以通过 Docker volume（参见范例 1.18 和范例 1.19）对数据进行管理。

(3) 使用 .dockerignore 文件。在镜像构建过程中，Docker 会将 Dockerfile 所在文件夹下的内容（即 build context）复制到构建环境中。使用 .dockerignore 文件可以将指定文件或者文件夹在镜像构建时从文件复制列表中排除。如果你不使用 .dockerignore 文件，请确保在只有所需最小集合的文件夹下构建镜像。请参考一下 .dockerignore 的语法（https://docs.docker.com/reference/builder/#dockerignore-file）。

(4) 利用 Docker Hub 的官方镜像（https://registry.hub.docker.com/search?q=library），而不是自己从头编写。这些镜像由软件的开发人员负责维护和保障。你也可以使用 `ONBUILD` 镜像（参见范例 2.10）来进一步简化你的镜像。

(5) 最后，最大限度地减少镜像层的数量，并利用镜像缓存的优点。Docker 使用联合文件系统存储镜像。这意味着每个镜像都由一个基础镜像和一组添加了必要修改的变更构成。每个变更都表示一个额外的镜像层。这对你编写 Dockerfile 以及如何使用各种指令会产生直接影响。随后的内容将会进一步对此进行说明。

2.5.3 讨论

在范例 2.4 中，你编写了自己的第一个 Dockerfile，包括以下指令。

```
FROM ubuntu:14.04

RUN apt-get update
```

```
RUN apt-get install -y python
RUN apt-get install -y python-pip
RUN apt-get clean

RUN pip install flask

ADD hello.py /tmp/hello.py
...
```

这个文件包含了一些错误，应该根据上面的最佳实践进行修改。

使用官方镜像 ubuntu:14.04 是一个好的做法。但是，在这之后你使用了多条 RUN 指令安装了一些软件包。这是一个不好的实践，它将会向镜像增加不必要的镜像层数。你也通过 ADD 指令复制了一个简单的文件。实际上，在这个例子里，更应该使用 COPY 指令（ADD 用于更复杂的文件复制场景）。

因此，这个 Dockerfile 应该像下面这样进行改写。

```
FROM ubuntu:14.04

RUN apt-get update && apt-get install -y \
    python
    python-pip
RUN pip install flask

COPY hello.py /tmp/hello.py
...
```

你还可以通过使用官方 Python 镜像，来进一步对这个 Dockerfile 进行优化，如下所示。

```
FROM python:2.7.10

RUN pip install flask

COPY hello.py /tmp/hello.py
...
```

当然你并不需要做到这种地步，这只是让你对如何优化 Dockerfile 有更深入的体会。要想获得更详细的信息，可以参考官方推荐的最佳实践（https://docs.docker.com/articles/dockerfile_best-practices/）。

2.6 通过标签对镜像进行版本管理

2.6.1 问题

你创建了多个镜像以及同一个镜像的多个版本。你希望通过采取某种方式来对镜像和镜像的版本进行跟踪，而不是使用镜像 ID。

2.6.2 解决方案

通过 `docker tag` 命令为镜像打标签。这允许你对已有镜像进行重命名，或者为同一个镜像名创建新的标签。

在通过提交容器创建镜像（参见范例 2.1）时你已经使用过标签了。镜像的命名规则会将冒号后面的所有内容看作镜像的标签。

> 镜像标签是可选的。如果你没有指定标签，Docker 会默认使用 `latest` 作为标签。如果指定镜像的标签在镜像仓库中不存在，那么下载 Docker 镜像将会失败。

举例来说，让我们将 ubuntu:14.04 镜像重命名为 foobar。这里没有指定标签，而只是修改了镜像名，因此 Docker 会自动使用 `latest` 标签。

```
$ docker images
REPOSITORY          TAG               IMAGE ID        CREATED          VIRTUAL SIZE
ubuntu              14.04             9bd07e480c5b    12 days ago      192.7 MB

$ docker tag ubuntu foobar
2014/12/17 09:57:48 Error response from daemon: No such id: ubuntu

$ docker tag ubuntu:14.04 foobar

$ docker images
REPOSITORY          TAG               IMAGE ID        CREATED          VIRTUAL SIZE
foobar              latest            9bd07e480c5b    12 days ago      192.7 MB
ubuntu              14.04             9bd07e480c5b    12 days ago      192.7 MB
```

在上面的例子中，你最先会看到的是，当你尝试给 ubuntu 镜像打标签时，Docker 抛出了错误。这是因为 ubuntu 镜像只有一个 14.04 标签，而没有 latest 标签。在你的第二条命令中，你通过冒号为这个镜像指定了本地已有的标签，打标签操作成功了。Docker 创建了一个新的名为 foobar 的镜像，并自动添加了 latest 标签。如果你在新的镜像名后面使用冒号为镜像设置一个标签，那么你会得到下面的结果。

```
$ docker tag ubuntu:14.04 foobar:cookbook

$ docker images
REPOSITORY          TAG               IMAGE ID        CREATED          VIRTUAL SIZE
foobar              cookbook          9bd07e480c5b    12 days ago      192.7 MB
foobar              latest            9bd07e480c5b    12 days ago      192.7 MB
ubuntu              14.04             9bd07e480c5b    12 days ago      192.7 MB
```

到目前为止，我们使用的所有镜像都在 Docker 宿主机本地。但是，如果你想通过 registry 来共享镜像，就需要为镜像设置合适的名称。特别是如果你将镜像上传到 Docker Hub（https://hub.docker.com），就必须遵循 USERNAME/NAME 的格式。当使用私有 registry 时，还需要指定 registry 的主机名、一个可选的用户名和镜像的名称（即 REGISTRYHOST/USERNAME/NAME）。当然，这时候你仍然可以使用标签（即 :TAG）。

2.6.3　讨论

正确地设置镜像标签是在 Docker Hub（参见范例 2.9）或私有 registry（参见范例 2.11）上共享镜像的重要组成部分。`docker tag` 命令的帮助信息很简洁，它显示了 Docker 镜像的命名规则，即如何指定正确的命名空间，可以是一个本地的镜像，或者在 Docker Hub 上，或者在私有 registry 上，如下所示。

```
$ docker tag -h

Usage: docker tag [OPTIONS] IMAGE[:TAG] [REGISTRYHOST/][USERNAME/]NAME[:TAG]

Tag an image into a repository

  -f, --force=false    Force
```

2.7　使用Docker provider从Vagrant迁移到Docker

2.7.1　问题

你一直在使用 Vagrant（http://vagrantup.com）来进行测试和开发，同时你想在 Docker 中重用一些你的 Vagrantfile 文件。

2.7.2　解决方案

使用 Vagrant Docker provider（https://docs.vagrantup.com/v2/docker/index.html）。你可以继续编写 Vagrantfile 文件来创建新的容器并编写你的 Dockerfile 文件。

下面是一个使用 Docker provider 的 Vagrantfile 文件示例。

```
# -*- mode: ruby -*-
# vi: set ft=ruby :

VAGRANTFILE_API_VERSION = "2"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|

  config.vm.provider "docker" do |d|
     d.build_dir = "."
  end

  config.vm.network "forwarded_port", guest: 5000, host: 5000

end
```

`build_dir` 选项指定了在与 Vagrantfile 相同的文件夹下面查找 Dockerfile 文件。之后 Vagrant 将会执行 `docker build` 命令并启动容器，如下所示。

```
$ vagrant up --provider=docker
Bringing machine 'default' up with 'docker' provider...
==> default: Building the container from a Dockerfile...
```

```
    default: Sending build context to Docker daemon 8.704 kB
    default: Step 0 : FROM ubuntu:14.04
...
==> default: Creating the container...
    default:   Name: provider_default_1421147689
    default:  Image: 324f2babf057
    default: Volume: /vagrant/provider:/vagrant
    default:   Port: 5000:5000
    default:
    default: Container created: efe111afb8b9d3ff
==> default: Starting container...
==> default: Provisioners will not be run since container doesn't support SSH.
```

在 `vagrant up` 命令执行完成之后，容器就会开始运行，并构建一个新的镜像。可以使用普通的 Docker 命令与这个容器进行交互，或者使用新的 `vagrant docker-logs` 和 `vagrant docker-run` 命令。标准的命令（比如 `vagrant status` 和 `vagrant destroy`）也可以与新的容器一起使用。

> 也许你并不希望在容器中安装 SSH。因此，Vagrant provisioner 不会运行。需要通过 Dockerfile 来安装任何想在容器中使用的软件。

2.7.3 讨论

为了帮助你实验该范例的内容，我创建了一个简单的实验环境。与其他范例一样，你可以克隆本书附带的 Git 仓库，并进入本范例的例子所在的文件夹。这个例子会启动一个 Ubuntu 14.04 的虚拟机，里面安装了 Docker 和 Vagrant。在 /vagrant/provider 文件夹下，你会发现另一个 Vagrantfile（在前面出现过），以及一个 Dockerfile。这个 Dockerfile 将会构建一个简单 Flask 应用的镜像，如下所示。

```
$ git clone
$ cd ch02/vagrantprovider/
$ vagrant up
$ vagrant ssh
$ cd /vagrant/provider
$ vagrant up --provider=docker
```

Vagrantfile 文件中可能的配置几乎与 Dockerfile 文件中的指令是一一对应的。你可以定义在容器中安装什么软件，传递什么环境变量，暴露哪些端口，链接到哪些容器，挂载哪些卷。有趣的是，Vagrant 会试图将常规的 Vagrant 配置转换为 Docker 的运行选项。例如，将 Docker 容器的端口转发到宿主机可以通过下面的 Vagrant 常规命令。

```
config.vm.network "forwarded_port", guest: 5000, host: 5000
```

总的来说，我的看法是，如果你已经在 Vagrant 中投入了大量的工作，并且愿意逐步过渡到 Docker，那么 Vagrant 对 Docker 的支持应该被视作一种过渡步骤。

> Vagrant 也提供了 Docker provisioner（https://docs.vagrantup.com/v2/provisioning/docker.html）。当你在启动虚拟机，通过配置管理工具（比如 Puppet 或 Chef）进行 provision 时，如果还想在虚拟机中启动容器，可以使用这个功能。

2.7.4 参考

- Vagrant Docker provider 配置（https://docs.vagrantup.com/v2/docker/configuration.html）
- Vagrant Docker provider 文档（https://docs.vagrantup.com/v2/docker/index.html）

2.8 使用Packer构建Docker镜像

2.8.1 问题

你通过 Chef（http://www.getchef.com）、Puppet（http://www.getchef.com）、Ansible（http://www.ansible.com/home）或 SaltStack（http://www.saltstack.com）开发了一些配置管理脚本。你希望使用这些配置脚本来构建 Docker 镜像。

2.8.2 解决方案

使用 HashiCorp 的 Packer（https://www.packer.io）。Packer 是一个基于单个模板定义为多个平台创建相同计算机镜像的工具。比如，你可以基于一个模板自动创建适用于 Amazon EC2（一个 AMI，即 Amazon Machine Image）、VMware、VirtualBox 和 DigitalOcean 的镜像。Docker 也是 Packer 所支持的一个平台。

也就是说，如果你创建了一个 Packer 模板，就可以自动生成一个 Docker 镜像。通过后置处理，你可以为镜像打标签，以及将镜像推送到 Docker Hub（参见范例 2.9）。

下面的模板展示了三个主要步骤。首先它指定了一个构建器，这里使用了 Docker 并且指定了基础镜像 ubuntu:14.04。其次，它定义了 provisioning 步骤。这里我们使用了一个简单的 shell 来进行 provisioning。最后是后置处理步骤，这里我们为构建的镜像打了标签。

```
{
"builders": [
  {
    "type": "docker",
    "image": "ubuntu:14.04",
    "commit": "true"
  }
],
"provisioners": [
  {
    "type": "shell",
    "script": "bootstrap.sh"
  }
],
```

```
"post-processors": [
   {
     "type": "docker-tag",
     "repository": "how2dock/packer",
     "tag": "latest"
   }
]
}
```

可以使用下面两个命令来验证模板的正确性，并构建镜像。

```
$ packer validate template.json
$ packer build template.json
```

> 你可以在模板中设置多个构建器，为你的应用分别输出不同类型的镜像（比如 Docker 和 AMI）。

为了帮助你试用 Packer，我创建了一个 Vagrantfile，它会启动一台 Ubuntu 14.04 虚拟机，并在其中安装 Docker，下载 Packer。你可以像下面这样使用这个 Vagrantfile。

```
$ git clone https://github.com/how2dock/docbook.git
$ cd ch02/packer
$ vagrant up
$ vagrant ssh
$ cd /vagrant
$ /home/vagrant/packer validate template.json
Template validated successfully.
$ /home/vagrant/packer build template.json
...
==> docker: Creating a temporary directory for sharing data...
==> docker: Pulling Docker image: ubuntu:14.04
...
==> Builds finished. The artifacts of successful builds are:
--> docker: Imported Docker image: 3ebae8e2f2a8af8f2c5f366c603091c5e9c8e234bff8
--> docker: Imported Docker image: how2dock/packer:latest
$ docker images
REPOSITORY          TAG         IMAGE ID        CREATED           VIRTUAL SIZE
how2dock/packer     latest      3ebae8e2f2a8    20 seconds ago    210.8 MB
ubuntu              14.04       8eaa4ff06b53    11 days ago       192.7 MB
```

在这个例子中，你可以直接运行 Nginx（Nginx 镜像已经通过 boostrap.sh 脚本安装）。

```
$ docker run -d -p 80:80 how2dock/packer /usr/sbin/nginx -g "daemon off;"
```

由于这个镜像不是通过 Dockerfile 构建的，因此 `CMD` 或 `ENTRYPOINT` 也都没有定义。当你启动容器时，Nginx 也不会自动开始运行。由于这个镜像在构建时没有指定如何运行 Nginx，所以基于这个镜像创建的容器在启动后会立即结束运行。

2.8.3　讨论

Packer 是一个非常不错的工具，它能帮助你从之前的 DevOps 工作流迁移到基于 Docker 的工作流。但是 Docker 容器需要以前台方式运行应用，并且推荐在一个容器中只运行一个应用进程。因此，使用 Packer 创建的镜像可能包括很多应用，比如 MySQL、Nginx 和 WordPress，这将违反 Docker 的理念，并且如果不额外使用类似 Supervisor（参见范例 1.15）这样的工具，你将很难运行多个进程。

上面的例子介绍了简单的基于 shell 的 provisioning 方式。如果你之前就有一些配置管理范例，那么在 Packer 中仍然可以使用这些范例来构建 Docker 镜像。Packer 支持 shell、Ansible、Chef、Puppet 和 Salt 等 provisioner（https://www.packer.io/docs/provisioners/shell.html）。举例来说，在前面我们使用的仓库中的 template-ansible.json 文件使用了 Ansible 本地 provisioner。修改后的 Packer 模板内容如下所示。

```
{
"builders": [
  {
    "type": "docker",
    "image": "ansible/ubuntu14.04-ansible:stable",
    "commit": "true"
  }
],
"provisioners": [
  {
    "type": "ansible-local",
    "playbook_file": "local.yml"
  }
],
"post-processors": [
   {
     "type": "docker-tag",
     "repository": "how2dock/packer",
     "tag": "ansible"
   }
]
}
```

它使用了一个从 Docker Hub 拉取的特殊 Docker 镜像，这个镜像安装了 Ansible。Packer 会使用本地的 Ansible CLI 来运行 Ansible 的 palybooklocal.yml。这个 playbook 在模板中定义，会在本地安装 Nginx，如下所示。

```
---
- hosts: localhost
  connection: local
  tasks:
   - name: install nginx
     apt: pkg=nginx state=installed update_cache=true
```

执行 Packer 的构建过程会得到一个可工作的镜像，镜像名为 how2dock/packer:ansible。

```
$ /home/vagrant/packer build template-ansible.json
...
```

```
==> docker: Creating a temporary directory for sharing data...
==> docker: Pulling Docker image: ansible/ubuntu14.04-ansible:stable
    docker: Pulling repository ansible/ubuntu14.04-ansible
...
==> docker: Provisioning with Ansible...
    docker: Creating Ansible staging directory...
    docker: Creating directory: /tmp/packer-provisioner-ansible-local
    docker: Uploading main Playbook file...
...
    docker:
    docker: PLAY [localhost] **************************************
    docker:
    docker: GATHERING FACTS ***************************************
    docker: ok: [localhost]
    docker:
    docker: TASK: [install nginx] *********************************
    docker: changed: [localhost]
    docker:
    docker: PLAY RECAP ********************************************
    docker: localhost  : ok=2    changed=1    unreachable=0    failed=0
    docker:
==> docker: Committing the container
..
    docker (docker-tag): Repository: how2dock/packer:ansible
Build 'docker' finished.
```

使用 Ansible playbook 构建的新镜像 how2dock/packer:ansible 已经可以使用了。你可以像前面的例子一样，从这个镜像创建一个新容器并在容器内运行应用程序。这一方法的有趣之处在于，使用容器代替虚拟机时，你还可以继续发挥配置管理工具中范例 /playbook 的价值。

2.9 将镜像发布到Docker Hub

2.9.1 问题

你编写了一个 Dockerfile 文件并构建了一个有用的镜像。你想和所有人共享这一镜像。

2.9.2 解决方案

你可以在 Docker Hub（http://hub.docker.com）上共享这个镜像。如果说 GitHub 是为源代码服务的，那么 Docker Hub 可以说是专门为 Docker 镜像服务的。Docker Hub 允许任何人在线托管他们的镜像，可以将镜像设为公开状态或者私有状态。为了在 Docker Hub 上共享镜像，你需要执行以下操作。

- 注册一个 Docker Hub 账号。
- 在你的 Docker 主机上登录 Docker Hub。
- 将你的镜像推送到 Docker Hub。

现在就让我们开始。注册 Docker Hub 账号只需要一个合法的电子邮件地址即可。打开注册页面（https://hub.docker.com/），注册一个账号。在完成了你注册账号时填写的邮箱地址

的验证之后，注册就宣告完成。你可以使用这个免费账号创建无限数量的公开镜像仓库和一个私有镜像仓库。如果希望创建不止一个私有仓库，你需要使用付费方案。

现在你已经拥有了一个 Docker Hub 账号。此时，你可以回到 Docker 主机，选择一个镜像，使用 Docker 命令行工具将这个镜像推送到你的公开仓库中。这需要执行以下三个步骤才能完成。

(1) 通过 `docker login` 命令来登录 Docker Hub，这会要求你输入 Docker Hub 凭据。
(2) 使用你 Docker Hub 上的用户名为已有镜像打标签。
(3) 将新打完标签的镜像推送到 Docker Hub。

登录过程会将你的 Docker Hub 凭据保存到文件 ~/.dockercfg 中，如下所示。

```
$ docker login
Username: how2dock
Password:
Email: how2dock@gmail.com
Login Succeeded
$ cat ~/.dockercfg
{"https://index.docker.io/v1/":{"auth":"..........",
                                "email":"how2dock@gmail.com"}}
```

如果查看一下当前你所拥有的镜像，你会看到在范例 2.4 中构建的 Flask 镜像使用了一个本地仓库名以及 `latest` 标签，如下所示。

```
$ docker images
REPOSITORY      TAG             IMAGE ID        CREATED         VIRTUAL SIZE
flask           latest          88d6464d1f42    5 days ago      354.6 MB
...
```

要想将这个镜像推送到你的 Docker Hub 账号下，你需要通过 `docker tag` 命令使用你在 Docker Hub 上的仓库为这个镜像打标签（参见范例 2.6），如下所示。

```
$ docker tag flask how2dock/flask
sebimac:flask sebgoa$ docker images
REPOSITORY          TAG         IMAGE ID            CREATED         VIRTUAL SIZE
flask               latest      88d6464d1f42        5 days ago      354.6 MB
how2dock/flask      latest      88d6464d1f42        5 days ago      354.6 MB
```

现在你的 Flask 镜像有了一个 how2dock/flask 的仓库名，这也符合 Docker Hub 的仓库命名规则。你已经可以推送镜像了。Docker 会推送这个组成镜像的各个镜像层；如果这个镜像层在 Docker Hub 上已经存在了，那么这个层就会被略过。在镜像推送完成之后，你就可以在你的 Docker Hub 页面上看到 how2dock/flask 镜像了，并且所有人都可以通过 `docker pull how2dock/flask` 来下载这个镜像（参见图 2-1），如下所示。

```
$ docker push how2dock/flask
The push refers to a repository [how2dock/flask] (len: 1)
Sending image list
Pushing repository how2dock/flask (1 tags)
511136ea3c5a: Image already pushed, skipping
01bf15a18638: Image already pushed, skipping
...
dc4a9a43bb7f: Image successfully pushed
```

```
e394b9fbe3fa: Image successfully pushed
3f7abcdc10d4: Image successfully pushed
88d6464d1f42: Image successfully pushed
Pushing tag for rev [88d6464d1f42] on
{https://cdn-registry-1.docker.io/v1/repositories/how2dock/flask/tags/latest}
```

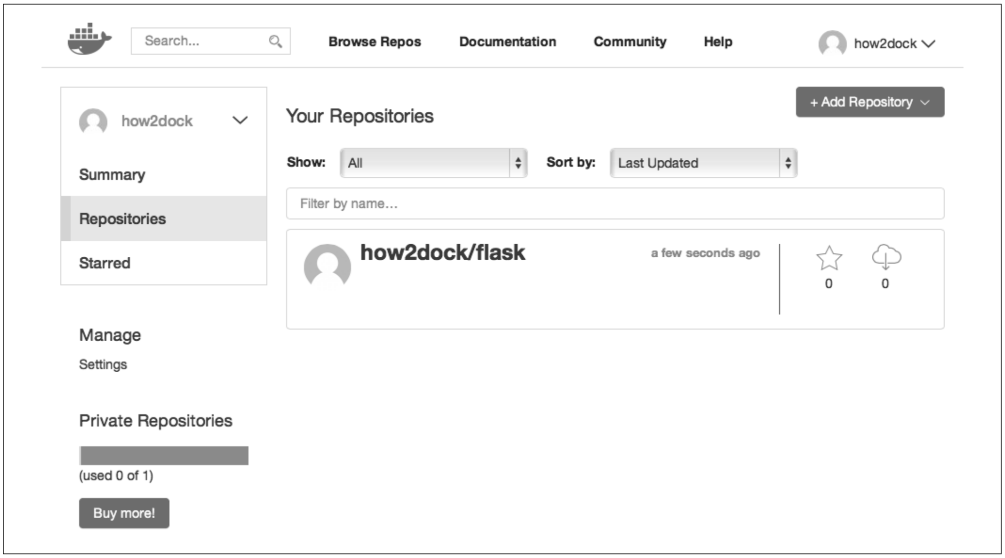


图 2-1：Docker Hub 上的 Flask 镜像

2.9.3 讨论

`docker tag` 命令可以用来修改镜像的仓库名和标签。在这例子中，你没有指定标签，所以 Docker 会自动设置一个 `latest` 标签。你可以选择为镜像设置标签并将标签信息推送到 Docker Hub 上，以此在同一仓库中维护一个镜像的多个版本。

本范例介绍了两个新的 Docker CLI 命令：`docker tag` 和 `docker push`。还有一个用于镜像管理的命令也值得关注：`docker search`。可以使用这条命令在 Docker Hub 上查找镜像。比如，可以像下面这样来查找提供 `postgres` 应用的镜像。

```
$ docker search postgres
NAME                      DESCRIPTION           STARS     OFFICIAL    AUTOMATED
postgres                  The PostgreSQL   ...  402       [OK]
paintedfox/postgresql     A docker image   ...  50                    [OK]
helmi03/docker-postgis    PostGIS 2.1 in   ...  20                    [OK]
atlassianfan/jira         Atlassian Jira   ...  17                    [OK]
orchardup/postgresql      https://github   ...  16                    [OK]
abevoelker/ruby           Ruby 2.1.2, Post ...  13                    [OK]
slafs/sentry              my approach for  ...  12                    [OK]
...
```

这条命令会返回超过 600 个镜像。第一条结果是由 Postgres 团队维护的官方镜像。一些镜像是自动构建并推送到 Docker Hub 的，你将会在范例 2.12 中学习如何使用自动构建镜像。

2.9.4 参考

- Docker Hub 指南（http://docs.docker.com/engine/userguide/dockerrepos/）

2.10 使用ONBUILD镜像

2.10.1 问题

你在一些软件仓库中看到有些 Dockerfile 文件中有一行类似 `FROM golang:1.3-onbuild` 的内容，你对这种镜像的工作原理很好奇。

2.10.2 解决方案

`ONBUILD` 指令为 Dockerfile 带来了一些小魔法。这条指令定义了一个会在未来执行的触发器。这个触发器是一个常规 Dockerfile 指令，比如 `RUN` 或 `ADD`。包含 `ONBUILD` 指令的镜像我们称之为父镜像。当一个父镜像被用作基础镜像时（即通过 `FROM` 指令被引用），新镜像也被称为子镜像，子镜像构建过程会触发在父镜像 `ONBUILD` 中定义的指令。

换句话说，就是父镜像会指定子镜像在构建时要做些什么。

你仍然可以在子镜像的 Dockerfile 文件中添加指令，但是父镜像中 `ONBUILD` 指令的内容会最先执行。

这非常方便构建极其简单的 Dockerfile，并能在所有的镜像之间提供一致性。

可以参考一下下面几个使用了 `ONBUILD` 指令的父镜像的例子。

- Node.js（https://github.com/nodejs/docker-node/blob/4654fb4bd58a36ae016a907c10b75daf9251a15d/0.12/onbuild/Dockerfile）
- Golang（https://github.com/docker-library/golang/blob/396f40c6188614c7acd6d8299a0ea71030a056a6/1.4/onbuild/Dockerfile）
- Python（https://github.com/docker-library/python/blob/7663560df7547e69d13b1b548675502f4e0917d1/2.7/onbuild/Dockerfile）
- Ruby（https://github.com/docker-library/ruby/blob/4ccabb5557ce2001aa1ae2a5f719340eb33c0383/2.1/onbuild/Dockerfile）

比如，对于 Node.js 应用来说，你可以使用下面 Dockerfile 定义的镜像作为父镜像。

```
FROM node:0.12.6

RUN mkdir -p /usr/src/app
WORKDIR /usr/src/app

ONBUILD COPY package.json /usr/src/app/
ONBUILD RUN npm install
ONBUILD COPY . /usr/src/app

CMD [ "npm", "start" ]
```

子镜像的 Dockerfile 看起来会像下面这样（最小指令集）。

```
FROM node:0.12.6-onbuild
```

在构建这个子镜像时，Docker 会自动将 package.json 文件从本地环境复制到 /usr/src/app 下面，然后运行 `npm install` 命令，将整个构建环境复制到 usr/src/app 目录。

2.10.3　参考

- 了解 `ONBUILD` 指令（http://www.eikonomega.com/dockerfile-understand-onbuild/）
- `ONBUILD` 文档（http://docs.docker.com/reference/builder/#onbuild）

2.11　运行私有registry

2.11.1　问题

使用 Docker Hub 非常简单。然而，你可能在数据治理方面比较关心将镜像托管在自己基础设施之外所带来的风险。因此，你希望在自己的基础设施之上运行自己的 Docker registry。

2.11.2　解决方案

使用 Docker registry 镜像（https://hub.docker.com/_/registry/）创建一个容器。这样，你就拥有了一个私有的 registry。

拉取 registry 镜像并以守护方式启动一个容器。然后，你可以通过 `curl` 访问 http://localhost:5000/v2 来确认一下 registry 是否正常运行，如下所示。

```
$ docker pull registry:2
$ docker run -d -p 5000:5000 registry:2
$ curl -i http://localhost:5000/v2/
HTTP/1.1 200 OK
Content-Length: 2
Content-Type: application/json; charset=utf-8
Docker-Distribution-Api-Version: registry/2.0
Date: Wed, 19 Aug 2015 23:07:47 GMT
```

上面的输出结果显示了 Docker registry 正在运行，其 API 版本为 v2。要想使用这个私有 registry，需要按照正确的命名规则，为你之前创建的本地镜像（比如在范例 2.4 中创建的 flask 镜像）打上标签。在我们的例子中，registry 运行在 http://localhost:5000 上，所以我们打标签的时候会使用 `localhost:5000` 作为前缀，并将这个镜像推送到私有 registry。也可以使用 Docker 主机的 IP 地址，如下所示。

```
$ docker tag busybox localhost:5000/busy
$ docker push localhost:5000/busy
The push refers to a repository [localhost:5000/busy] (len: 1)
8c2e06607696: Image successfully pushed
6ce2e90b0bc7: Image successfully pushed
cf2616975b4a: Image already exists
latest: digest: sha256:3b5b980...a4d59f24f9c7253fce29 size: 5049
```

如果从其他计算机访问这个私有 registry，你会收到一个错误消息，提示你的 Docker 客户不能使用一个不安全的 registry。如果是测试环境（生产环境不建议这样操作），可以编辑你的 Docker 配置文件，增加 `insecure-registry` 选项。比如，在 Ubuntu 14.04 上编辑 /etc/default/docker 文件，添加如下一行。

```
DOCKER_OPTS="--insecure-registry <IP_OF_REGISTRY>:5000"
```

重新启动 Docker 服务（`sudo service docker restart`），然后再次访问远程私有 registry。（记住，需要在 registry 所在计算机之外的其他计算机上进行上述操作。）

2.11.3　讨论

这个简短的例子使用了 registry 的默认设置。默认设置下，不需要进行身份验证，registry 是不安全的，会使用本地存储方式，并会使用一个 SQLAlchemy 搜索引擎。这些选项都可以通过设置环境变量或者编辑配置文件来修改。这些在官方文档（https://github.com/docker/distribution）中都有详细的描述。

通过 Docker 镜像运行的 registry 是一个使用 Golang 编写的应用程序，它对外提供了一套 HTTP REST API（https://docs.docker.com/registry/spec/api/）。你可以使用自己的 Docker 客户端甚至是 `curl` 来访问这个 registry。

比如，要想列出私有 registry 保存的所有镜像，可以使用 /v2/_catalog URI，如下所示。


```
$ curl http://localhost:5000/v2/_catalog
{"repositories":["busy"]}
```

如果以不同的标签将 busybox 镜像推送到私有 registry，你将会在返回的镜像列表中看到这个新添加的镜像，如下所示。

```
$ docker tag busybox localhost:5000/busy1
$ docker push localhost:5000/busy1
...
$ curl http://localhost:5000/v2/_catalog
{"repositories":["busy","busy1"]}
```

每个 registry 中的镜像都使用清单对镜像进行描述。可以通过 /v2//manifests/ 这个 API 来获得镜像的清单描述，其中 `<name>` 是镜像的名称，而 `<reference>` 是镜像的标签。

```
$ curl http://localhost:5000/v2/busy1/manifests/latest
{
   "schemaVersion": 1,
   "name": "busy1",
   "tag": "latest",
   "architecture": "amd64",
   "fsLayers": [
      {
         "blobSum": "sha256:a3ed95caeb02ffe68cdd9fd84406680ae93d633cb16422d..."
      },
      {
         "blobSum": "sha256:1db09adb5ddd7f1a07b6d585a7db747a51c7bd17418d47e..."
      },
```

```
        {
          "blobSum": "sha256:a3ed95caeb02ffe68cdd9fd84406680ae93d633cb16422d..."
        }
      ],
   ...
```

上面看到的每个 blob 都代表一个镜像层。可以通过 registry API 来进行上传、下载或删除 blob 操作。registry API 规范文档页面（https://docs.docker.com/registry/spec/api/#deleting-an-image）中有详细的介绍。

要想列出一个镜像的所有标签，可以使用 v2//tags/listURI，如下所示。

```
$ curl http://localhost:5000/v2/busy1/tags/list
{"name":"busy1","tags":["latest"]}
$ docker tag busybox localhost:5000/busy1:foobar
$ docker push localhost:5000/busy1:foobar
$ curl http://localhost:5000/v2/busy1/tags/list
{"name":"busy1","tags":["foobar","latest"]}
```

这些例子都使用了 `curl`，主要是为了让你对 registry API 有一个更直观的感受。完整的 API 文档可以在 Docker 的官方网站（https://docs.docker.com/reference/api/registry_api/#set-a-tag-for-a-specified-image-id）上找到。

2.11.4　参考

- Docker Hub 上的 Docker registry 主页（https://hub.docker.com/）
- GitHub 上更丰富的文档（https://github.com/docker/distribution）
- registry 部署说明（https://docs.docker.com/registry/deploying/）

2.12　为持续集成/部署在Docker Hub上配置自动构建

2.12.1　问题

你已经开始使用 Docker Hub（https://hub.docker.com，参见范例 2.9），并且已经向 Docker Hub 推送了镜像，但都是通过手动方式进行的。你希望在每次提交对镜像的修改时都能自动构建镜像。

2.12.2　解决方案

不要设置标准仓库，而是创建自动构建仓库，并指向 GitHub 或 Bitbucket 上的应用。

在 Docker Hub 页面，单击 Add Repository 按钮并选择 Automated Build（图 2-2）。然后就可以选择使用 GitHub 或者 Bitbucket（图 2-3）。

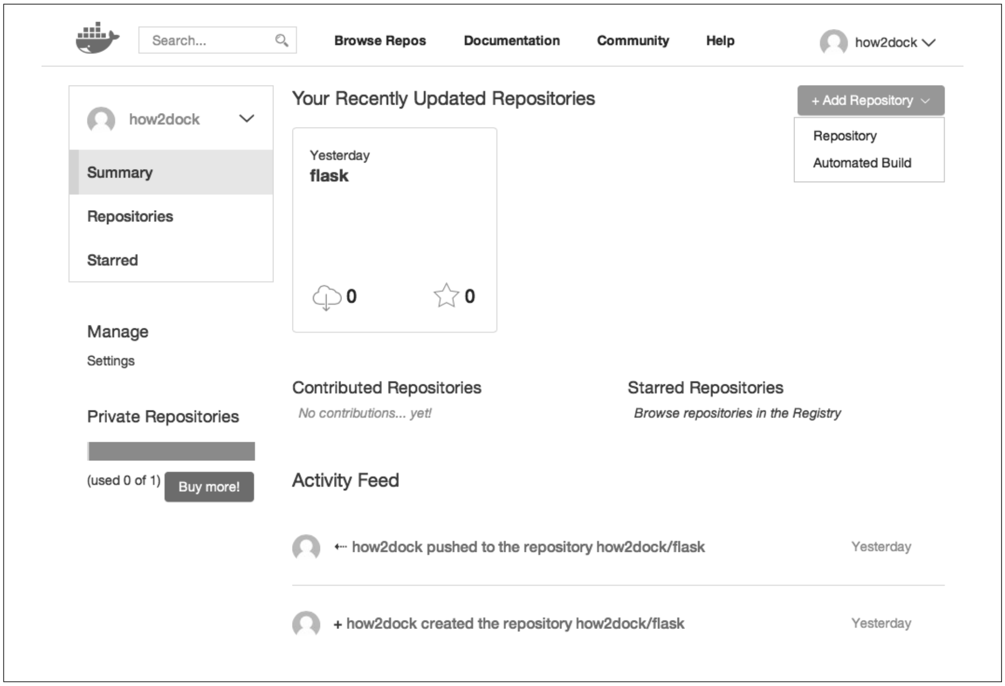


图 2-2：创建自动构建仓库

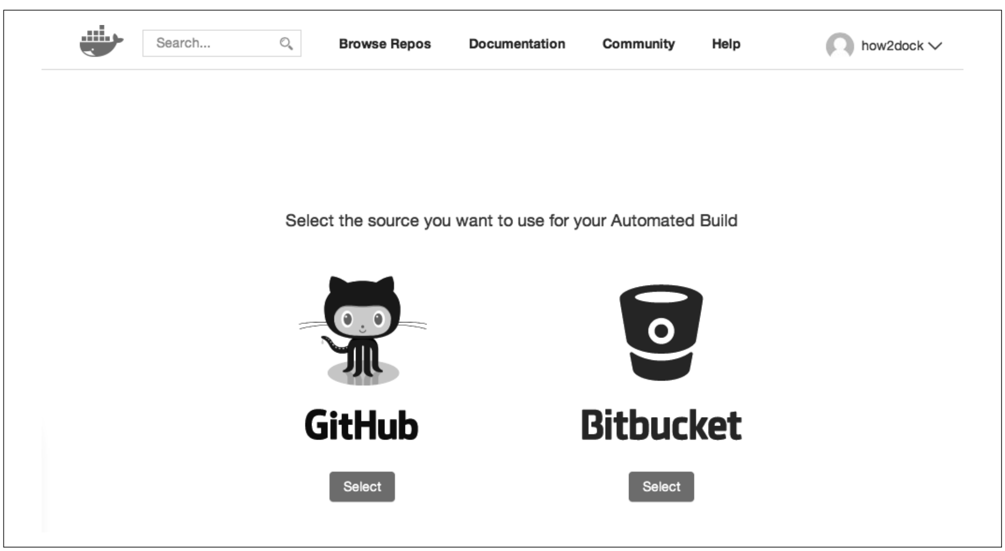


图 2-3：选择使用 GitHub 或 Bitbucket

你可以在 Docker Hub 上创建一个公开或者私有的自动构建仓库，并使用公开或者私有的代码仓库。如果你要创建一个私有的自动构建，Docker Hub 需要你具有 GitHub 账号的读写权限。

选择完要使用的在线版本控制系统之后，需要选择用来构建镜像的项目（图 2-4）。这应该是 GitHub 或者 Bitbucket 中有 Dockerfile 文件的项目，你想为这个项目自动构建 Docker 镜像。然后你可以设置一个 Docker Hub 仓库的名称，选择源代码所在分支，指定 Dockerfile 所在文件路径。这非常适合想在一个 GitHub 或者 Bitbucket 仓库中维护多个 Dockerfile 的情况。Docker Hub 会在你的 GitHub 仓库中创建一个 GitHub 钩子。

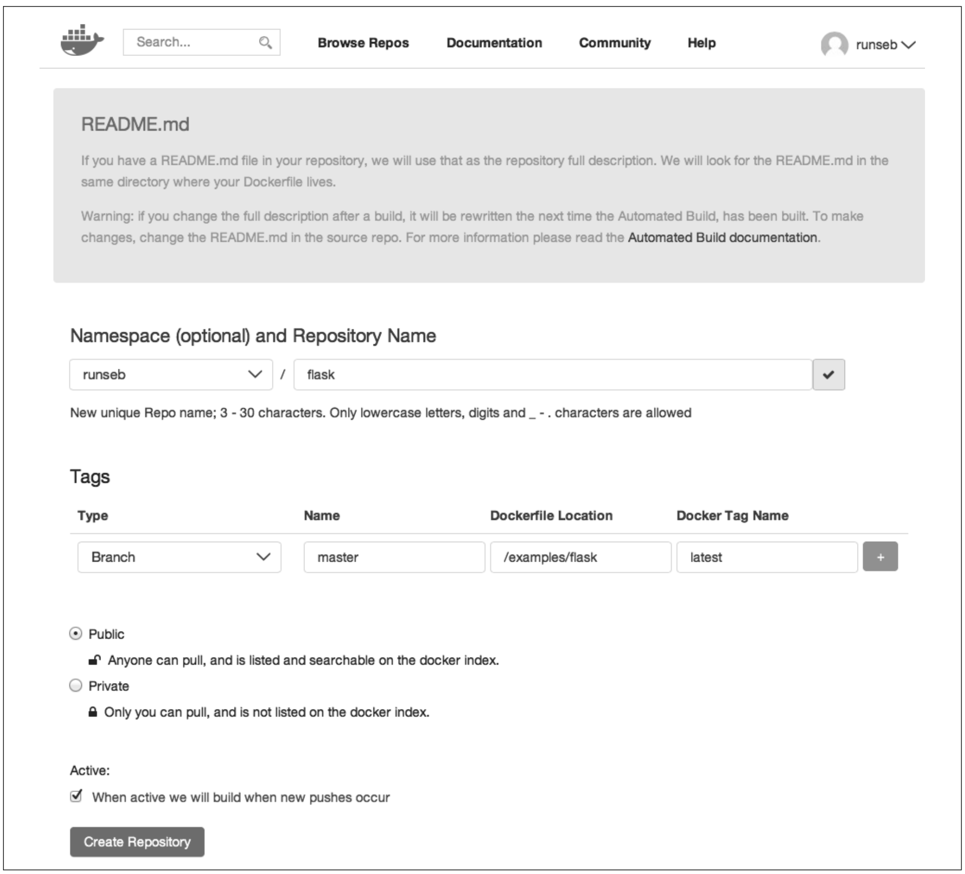


图 2-4：输入自动构建的详细信息

在 Docker Hub 上创建的镜像仓库名并不一定要与 GitHub/Bitbucket 上的仓库名保持一致。

当完成自动构建的配置之后，你就可以查看到构建的详情。构建的状态会从 pending 到 building 再到 pushing，最后会变为 finished。构建完成之后，你就可以拉取新镜像了。

```
$ docker pull runseb/flask
```

Dockerfile 选项卡会根据你的 GitHub 仓库中的 Dockerfile 文件的内容自动生成。如果存在 README.md 文件，则会自动根据 README.md 文件生成 Information 选项卡的内容。

向 GitHub 仓库推送了新的提交之后，Docker Hub 会自动触发一次镜像构建。当构建完成之后，新的镜像就可以使用了。

> 你也可以修改构建设置，从不同的分支触发构建，并为镜像设置不同的标签。比如，你可以从主分支进行构建并设置相关的标签为 `latest`，使用发布分支进行构建并设定一个不同标签（比如，为从 1.0 分支构建的镜像设置 `1.0` 的标签）。

2.12.3 讨论

除了通过向 GitHub 或者 Bitbucket 推送代码来自动触发构建，也可以通过发送一个 HTTP 的 POST 请求到指定的 URL 来触发构建，这个 URL 可以在 Build Trigger 页面得到（图 2-5）。为了防止对 Docker Hub 造成滥用，有些构建可能会被忽略。

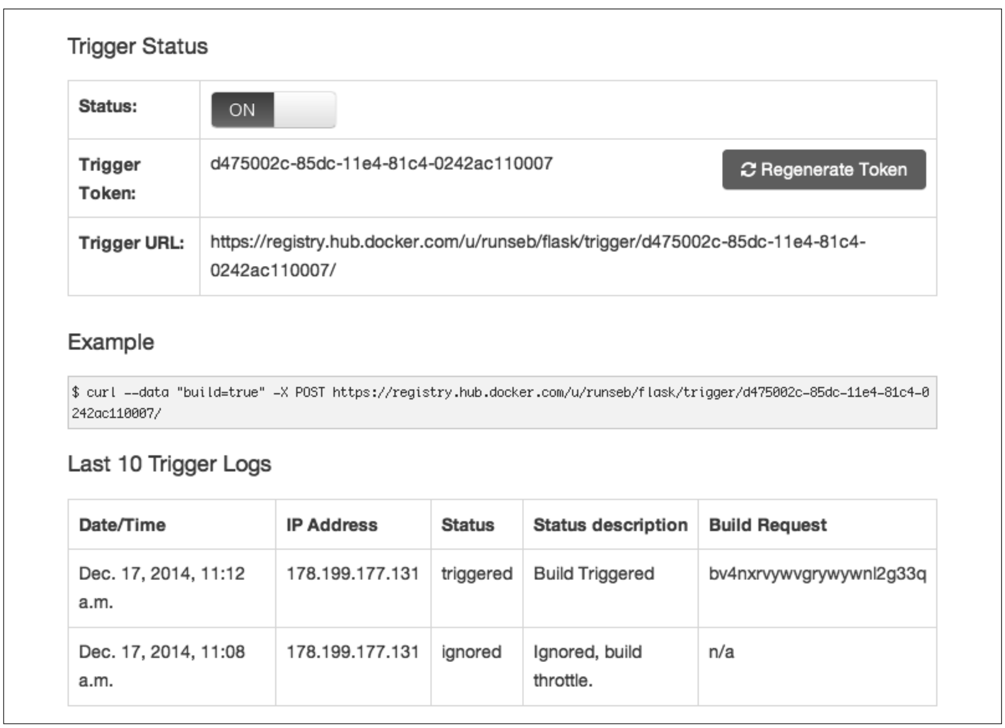


图 2-5：打开构建触发器

最后，不管你是使用自动构建还是使用触发器手动触发构建，都可以使用 webhook。webhook URL 非常适合与其他系统进行集成，比如 Jenkins（https://jenkins-ci.org）。有多种工具可以用来触发镜像构建以及作为其中的一步集成到持续交付工作流中。在自动构建的构建详情页面，你可以看到 webhook 页面。在这个页面，当构建成功时，你可以向指定的 URL 发送 HTTP POST 请求。这个请求体中包括一个回调用的 URL。你的接收网址需要发送一个 HTTP POST 请求作为返回结果，返回体是一个 JSON 数据，包括用于表示返回结果的 `state` 属性，它的值为 `success`、`failure` 或者 `error` 之一。当收到一个成功的状态时，自动构建就会继续调用下一个 webhook，这样就可以将多个操作串联到一起。

2.12.4　参考

- 自动构建参考文档（https://docs.docker.com/docker-hub/builds/）

2.13　使用Git钩子和私有registry建立本地自动构建环境

2.13.1　问题

使用 Docker Hub、GitHub 或者 Bitbucket 进行自动构建非常实用（参见范例 2.12），但是你可能正在使用私有 registry（比如一个本地的 hub），并且希望在向本地的 Git 项目中推送代码时触发 Docker 镜像构建。

2.13.2　解决方案

创建一个 Git 的 `post-commit` 钩子，由它来触发一个构建并将新镜像推送到你的私有 registry。

在你 Git 项目的根文件夹下创建一个 bash 脚本 ./git/hooks/post-commit，它的内容比较简单，如下所示。

```
#!/bin/bash

tag=`git log -1 HEAD --format="%h"`
docker build -t flask:$tag /home/sebgoa/docbook/examples/flask
```

使用 `chmod +x .git/hooks/post-commit` 命令将文件的属性设置为可执行。

现在，每当你向 Git 项目中提交代码，bash 脚本 post-commit 都会被执行。它将会使用提交 SHA 的简短散列字符串作为新的 `tag`，并使用指定 Dockerfile 文件触发一次构建。之后它就会构建一个新的名为 flask 的镜像，并使用由程序生成的标签。

```
$ git commit -m "fixing hook"
9c38962
Sending build context to Docker daemon 3.584 kB
Sending build context to Docker daemon
Step 0 : FROM ubuntu:14.04
 ---> 9bd07e480c5b
```

```
Step 1 : RUN apt-get update
 ---> Using cache
 ---> e659c9e9ba21
<snip>
Removing intermediate container 05c13744c7bf
Step 8 : CMD python /tmp/hello.py
 ---> Running in 124cd2ada52d
 ---> 9a50c7b2bee9
Removing intermediate container 124cd2ada52d
Successfully built 9a50c7b2bee9
[master 9c38962] fixing hook
 1 file changed, 1 insertion(+), 1 deletion(-)
$ docker images
REPOSITORY         TAG              IMAGE ID         CREATED          VIRTUAL SIZE
flask              9c38962          9a50c7b2bee9     5 days ago       354.6 MB
```

尽管上面的方法能正常工作，并且它只使用了两行 bash 代码，但是如果这个构建过程需要花费很长时间，那么在 Git 的 post-commit 任务中进行镜像构建可能就不太切合实际了。比较好的方法是使用 post-commit 钩子触发一个远程的构建，然后将新镜像推送到私有 registry。

2.13.3 讨论

举例来说，你可以使用 Git 钩子触发一个位于一台 Jenkins 服务器上的镜像构建任务，然后让 Jenkins 将新镜像推送到你的私有 registry。

2.14 使用Conduit进行持续部署

2.14.1 问题

你已经掌握了如何在 Docker Hub 上设置自动构建（参见范例 2.12），但是你想添加一个钩子，以便在镜像构建完成时，自动将新镜像部署到 Docker 主机上。

2.14.2 解决方案

Docker Hub 提供了 webhook 功能（https://docs.docker.com/docker-hub/builds/#webhooks），当你成功向 Docker Hub 进行推送时，它就会被调用。

webhook 是一个发送给指定回调地址的 HTTP POST 请求。通过在回调地址中对这个 HTTP 请求进行处理，解析请求内容，你可以拉取镜像或者启动新容器。Docker Hub 允许你将多个 webhook 串联起来，连续触发多个事件。

将 webhook 串接起来可以用于构建持续集成和持续部署工作流。一个开发团队会修改应用程序的代码（比如在 GitHub 上）。如果源代码中包括一个 Dockerfile，并且设置了自动构建，那么每次提交都会构建一个新的镜像。为了验证这个镜像，团队一般会运行集成测试。有很多可选的网络服务［包括 Travis CI（https://travis-ci.org）、CircleCI（https://circleci.com）和 Codeship（https://codeship.com）］可以帮你完成测试工作，你也可以自己通过 Jenkins 或者其他测试框架来完成测试工作。当镜像验证完成之后，就可以通过第二

个 webhook 在生产环境中自动部署这个新的镜像。

为了测试这一功能，你需要在 Docker Hub 上创建一个 webhook。这个 webhook 会访问一个应用程序，这个应用会处理 webhook 的请求内容并部署新镜像。Conduit 就是这样一个应用，并且可以在 Docker Hub 上找到。

> Conduit 还处于实验阶段，不建议在生产环境下使用。

2.14.3　参考

- Docker Hub webhook 官方文档（https://docs.docker.com/docker-hub/builds/#webhooks）
- Conduit 在 Docker Hub 上的主页（https://registry.hub.docker.com/u/ehazlett/conduit/）
- Conduit 在 GitHub 上的主页（https://github.com/ehazlett/conduit）

第3章

Docker网络

3.0 简介

当构建分布式应用程序时，组成它的各个服务需要能够互相通信。这些在容器中运行的服务，可能会运行在一台主机或多台主机上，甚至是跨越数据中心的不同主机上。因此，容器网络是任何基于 Docker 的分布式应用程序需要考虑的关键因素。

用于容器的网络技术与用于虚拟机的网络技术非常类似。在同一台主机上的容器可以连接到一个软件交换机上，iptables 可以用来控制容器之间的网络流量，并将在容器中运行的进程的端口暴露到宿主机上。

在安装时，Docker 引擎会在你的主机上对网络进行很多默认设置，这样一来，你不用做任何操作就可以直接使用 Docker 网络。在范例 3.1 中，我们首先将介绍一些 Docker 命令，你可以用这些命令获得容器的 IP 地址。之后在范例 3.2 中，我们将展示如何将容器中的端口映射到宿主机的端口上。在范例 3.3 中，我们将会更深入地了解一下容器链接技术，这是一种可以帮助多个容器进行服务发现的机制。

对于一个分布式应用，网络是非常重要的，我们认为有必要对其进行深入的研究。在范例 3.4 中，我们会介绍 Docker 默认的桥接配置。在范例 3.6 中，我们会向你展示如何通过修改 Docker 引擎的启动选项来更改默认值。范例 3.8 和范例 3.9 将会带你进行更深入的研究，并告诉你如何创建自己的网络交换机，使用它为你的容器提供网络通信。虽然不是必需的，但了解容器网络的工作原理并能对容器的网络进行配置，将会对在生产环境下运维自己的应用很有帮助。

虽然容器的网络和虚拟机的网络非常类似，但它们有一个主要的区别。在容器中，你可以选择使用哪个网络协议栈（参见范例 3.5）。比如，你可以让容器和宿主机共享网络协议

栈，这样你的容器就可以拥有与宿主机一样的 IP 地址。当然，你也可以让多个容器共享同一个网络协议栈。有多种容器网络模式可供选择，为了探索所有的可能性，范例 3.7 会介绍一个非常实用的工具：Pipework。花一点点时间学习一下 Pipework，了解它的功能，将会大大加深你对容器和网络的理解。

目前为止，所有的范例都是以一台主机为前提进行介绍的。然而在现实中，分布式应用程序可能涉及几十台、几百台甚至上千台主机。在范例 3.10 中，我们将展示一个基本的例子，在两台不同的主机之间建立一条隧道，为在两台主机上运行的容器提供一个通用的 IP 子网以进行通信。这个范例只用于教学目的，不能作为生产环境的解决方案。范例 3.11、范例 3.12 和范例 3.13 则会介绍 Weave 和 Flannel。这几个范例是由 Fintan Ryan 和 Eugene Yakubovich 提供的，介绍了多主机容器网络解决方案，这些方案可用于生产环境。

在本章结尾，我们将窥探一下 Docker Network（范例 3.14），并深入了解一下 VXLAN 的配置（范例 3.15）。Docker Network 目前正在开发中，但应该很快就能成为标准 Docker 发布的一部分。类似于 Weave、Flannel 和 Calico 这样的解决方案也能通过正在开发中的插件机制在 Docker Network 中使用。

现概述一下你将在本章学到什么内容。最初的几个范例会涉及一些基于 Docker 默认网络配置的基本概念。对开发人员来说，了解这些就已经足够了。关心在生产环境下部署基于 Docker 的应用程序的系统管理员，应该深入地研究一下 Docker 引擎的网络配置，更好地理解这些默认配置，以及网络命名空间在 Docker 中是如何使用的。最后，对于生产环境下的跨主机方案，你应该查看一下关于 Weave 和 Flannel 的范例，同时学习一下 Docker Network。

3.1　查看容器的IP地址

3.1.1　问题

你启动了一个容器，想知道这个容器的 IP 地址。

3.1.2　解决方案

在默认的 Docker 网络模式下，有很多方法可以获取一个容器的 IP 地址。本范例中将会介绍其中几种。

第一种方法就是使用 `docker inspect` 命令（参见范例 9.1 了解更多）并指定一个 Go 模板格式，如下所示。

```
$ docker run -d --name nginx nginx
$ docker inspect --format '{{ .NetworkSettings.IPAddress }}' nginx
172.17.0.2
```

也可以使用 `docker exec` 命令在容器内部执行命令来获得容器的 IP 地址，如下所示。

```
$ docker exec -ti nginx ip add | grep global
    inet 172.17.0.2/16 scope global eth0
```

假设 Docker 已经为这个容器设置好，还可以查看容器内的 /etc/hosts 文件，如下所示。

```
$ docker run -d --name foobar -h foobar busybox sleep 300
$ docker exec -ti foobar cat /etc/hosts | grep foobar
172.17.0.4      foobar
```

最后，可以进入容器中的 shell，输入标准的 Linux 命令，如下所示。

```
$ docker exec -ti nginx bash
root@a3c1f7edb00a:/# cat /etc/hosts
```

3.1.3 参考

- 为了更好地理解 Docker 网络的工作原理，了解除了获取容器 IP 地址以外的知识，请参见范例 3.4
- 10 个获取 Docker 容器 IP 地址的例子（http://networkstatic.net/10-examples-of-how-to-get-docker-container-ip-address/）

3.2 将容器端口暴露到主机上

3.2.1 问题

你希望在网络上访问运行在容器内的服务。

3.2.2 解决方案

Docker 可以通过 `docker run` 命令的 `-P` 选项将容器内的端口动态绑定到宿主机上。也可以通过 `-p` 选项手动指定映射关系。

假设你已经通过如下所示的 Dockerfile 构建了一个运行 Python Flask 应用的镜像。

```
FROM python:2.7.10

RUN pip install flask
COPY hello.py /tmp/hello.py

CMD ["python","/tmp/hello.py"]
```

这个文件与范例 2.4 中看到的有点类似。让我们构建这个镜像，并且不指定任何端口映射选项来创建一个容器，如下所示。

```
$ docker build -t flask
$ docker run -d --name foobar flask
```

你可以找出容器的 IP 地址，在宿主机上通过 5000 端口访问到容器内的 Flask 应用。

```
$ docker inspect -f '{{.NetworkSettings.IPAddress}}' foobar
172.17.0.2
$ curl http://172.17.0.2:5000/
Hello World!
```

但是，你不能在该容器所在宿主机之外通过这种方式来访问该应用。要想从外部也能访问该应用，你需要在启动容器时使用端口映射，如下所示。

```
$ docker kill foobar
$ docker rm foobar
$ docker run -d -p 5000 --name foobar flask
$ docker ps
CONTAINER ID  IMAGE   COMMAND             ... PORTS                       NAMES
2cc258827b34  flask   "python /tmp/hello.p ... 0.0.0.0:32768->5000/tcp   foobar
```

可以看到在 `docker ps` 命令的结果中，`PORTS` 列显示了 32768 端口和容器的 5000 端口之间的映射关系。宿主机会监听 0.0.0.0 网络接口以及 TCP 32768 端口，并将请求转发到容器的 5000 端口。如果在命令行上通过 `curl` 访问 Docker 主机的 32768 端口，就可以访问到容器内的 Flask 应用了。

> 除了 `docker ps` 命令会返回端口映射关系，以及你可以使用 `docker inspect`，Docker 还提供了一个很有用的 `docker port` 命令来显示容器的端口映射信息。比如，试试下面这样：
>
> ```
> $ docker port foobar 5000
> 0.0.0.0:32768
> ```

可能你已经注意到了，与范例 2.4 不同，上面的 Dockerfile 中并没有 `EXPOSE` 语句。如果你添加了该语句，就可以通过 `-P` 来暴露该端口而免去详细指定应用程序端口的麻烦。Docker 会自动进行正确的端口映射。在前面的 Dockerfile 中添加 `EXPOSE 5000`，然后重新构建镜像，像下面这样启动容器。

```
$ docker run -d -P flask
```

你会看到端口映射都是自动完成的。

3.2.3 讨论

一个容器可以同时暴露多个容器端口，也可以选择是使用 TCP 还是 UDP 协议。比如，你想将 5000 端口以 TCP 协议、53 端口以 UDP（假设你的应用也使用了 53 端口）协议暴露给外部，可以像下面这样操作。

```
$ docker run -d -p 5000/tcp -p 53/udp flask
```

端口映射通过两种机制实现。

首先，默认情况下 Docker 会修改宿主机的 iptables 规则。如果你在 Flask 应用运行时查看一下 iptables 的规则，就会发现在 Docker 链表中新增加的一条规则。

```
$ sudo iptables -L
...

Chain DOCKER (1 references)
target     prot opt source               destination
ACCEPT     tcp  --  anywhere             172.17.0.2           tcp dpt:5000
```

其次，Docker 会在宿主机上启动一个轻量的代理程序。这个进程监听宿主机的所有网络接口，监听端口是为容器动态分配的端口。查看系统正在运行的进程，就能看到这个代理程序。

```
$ ps -ef | grep docker
root     29969     1  ... /usr/bin/docker -d
root     30851 29969  ... docker-proxy -proto tcp -host-ip 0.0.0.0 \
                                                  -host-port 32769 \
                                                  -container-ip 172.17.0.5 \
                                                  -container-port 5000
```

可以修改这些默认动作来禁止 Docker 修改你的 iptables，这时你必须自己处理网络相关功能（参见范例 3.6）。

3.3 在Docker中进行容器链接

3.3.1 问题

当构建一个由多个服务构成的分布式应用时，你需要一种机制来发现这些服务的位置，这样系统中的各个组件才能互相通信。你可以手动为这些服务（运行于容器之中）设置 IP 地址，但是考虑到可扩展性，你需要一种自发现机制。

3.3.2 解决方案

在 Docker 中最容易想到的解决方案是将容器链接起来。这可以通过 `docker run` 命令的 `--link` 选项来实现。

> 容器链接在单台主机上工作良好，但是在一个大规模系统中，需要使用其他的服务发现方式。类似范例 7.13 这样再加上键值存储和 DNS 的解决方案也是一个不错的选择。Docker Network（参见范例 3.14）提供了一种内建机制来将容器内的服务暴露给外部，而不必使用容器链接。

为了解释容器链接，让我们构建一个由数据库、Web 应用和负载平衡组成的三层架构系统。首先启动一个数据库容器，然后将它链接到 Web 应用容器，最后再启动负载平衡容器，并链接 Web 应用容器。为了方便进行链接，需要为每个容器设置一个容器名。由于这是一个简单的例子，该应用程序不需要做任何工作，所以我们选择了使用 runseb/hostname 这个镜像。这是一个 Flask 应用，它会返回该应用所在容器的主机名。

让我们启动这三个容器，如下所示。

```
$ docker run -d --name database -e MYSQL_ROOT_PASSWORD=root mysql
$ docker run -d --link database:db --name web runseb/hostname
$ docker run -d --link web:application --name lb nginx
$ docker ps
CONTAINER ID  IMAGE          COMMAND              ... PORTS             NAMES
507edee2bbcf  nginx          "nginx -g 'daemon of ... 80/tcp, 443/tcp  lb
```

```
62c321acb102  runseb/hostname "python /tmp/hello   ... 5000/tcp         web
cf17b64e7017  mysql           "/entrypoint.sh mysq ... 3306/tcp         database
```

将容器链接起来之后，在 Web 应用容器中会包含一些指向数据库的环境变量。类似地，在负载平衡容器中，也存在指向 Web 应用容器的环境变量，如下所示。

```
$ docker exec -ti web env | grep DB
DB_PORT=tcp://172.17.0.13:3306
DB_PORT_3306_TCP=tcp://172.17.0.13:3306
DB_PORT_3306_TCP_ADDR=172.17.0.13
DB_PORT_3306_TCP_PORT=3306
DB_PORT_3306_TCP_PROTO=tcp
DB_NAME=/web/db
DB_ENV_MYSQL_ROOT_PASSWORD=root
DB_ENV_MYSQL_MAJOR=5.6
DB_ENV_MYSQL_VERSION=5.6.25

$ docker exec -ti lb env | grep APPLICATION
APPLICATION_PORT=tcp://172.17.0.14:5000
APPLICATION_PORT_8080_TCP=tcp://172.17.0.14:5000
APPLICATION_PORT_8080_TCP_ADDR=172.17.0.14
APPLICATION_PORT_8080_TCP_PORT=5000
APPLICATION_PORT_8080_TCP_PROTO=tcp
APPLICATION_NAME=/lb/application
```

可以用这些环境变量来动态配置应用程序和负载平衡。

/etc/hosts 文件也会自动更新，该文件包含了最新的名称解析相关信息。

```
$ docker exec -ti web cat /etc/hosts
172.17.0.14     62c321acb102
172.17.0.13     db cf17b64e7017 database

$ docker exec -ti lb cat /etc/hosts
172.17.0.15     507edee2bbcf
172.17.0.14     application 62c321acb102 web
```

如果你重启了一个容器，那么链接了这个容器的 /etc/hosts 文件会自动更新，而其环境变量则保持不变。因此，推荐使用 /etc/hosts 文件对所链接的容器进行 IP 地址解析。

3.3.3 讨论

当你启动容器时，你会发现，我们已经为每个容器都设置了名称。容器的名称可以用来定义容器链接。其格式如下所示。

```
--link <container_name>:<alias>
```

你会在 /etc/host 文件中看到为链接指定的别名（alias），为容器链接添加的环境变量也会以这个别名为前缀。

如果容器正在运行中，可以使用 `inspect` 方法来查看一下它都使用了哪些容器链接。返回结果为被链接的容器名与其在该容器中所用别名的映射关系。

```
$ docker inspect -f "{{.HostConfig.Links}}" application
[/database:/application/db]
$ docker inspect -f "{{.HostConfig.Links}}" lb
[/application:/lb/app]
```

尽管容器链接对于在单台主机上进行开发非常有用，但是在大规模的部署中也有其局限性，比如容器重新启动非常频繁。基于 DNS 或动态容器注册机制的系统具备扩展性和自动更新功能。

3.3.4 参考

- Docker 容器链接官方文档（https://docs.docker.com/userguide/dockerlinks/）

3.4 理解Docker容器网络

3.4.1 问题

你想了解 Docker 容器网络的基本知识。

3.4.2 解决方案

如果只是为了使用 Docker 并且让 Docker 容器能够互相通信，就并不一定要使用该解决方案。这里之所以对此进行详细介绍，就是希望为你提供更多的信息，以便你根据自己的喜好对 Docker 网络进行定制。

在默认安装情况下，Docker 会在宿主机上创建一个名为 `docker0` 的 Linux 网桥设备。该网桥设备拥有一个私有网络地址及其所属子网。分配给 `docker0` 网桥的子网地址为 172.[17-31].42.1/16、10.[0-255].42.1/16 和 192.168.[42-44].1/24 中第一个没有被占用的子网地址。因此，很多时候你的 `docker0` 网桥设备的地址都是 172.17.42.1。所有容器都会连接到该网桥设备上，并从中分配一个位于子网 172.17.42.0/24 中的 IP 地址。容器链接到网桥的网络接口会把 `docker0` 网络设备作为网关。创建新容器时，Docker 会创建一对网络设备接口，并将它们放到两个独立的网络命名空间：一个网络设备放到容器的网络命名空间（即 `eth0`）；另一个网络设备放到宿主机的网络命名空间，并连接到 `docker0` 网桥设备上。

为了说明这一创建过程，让我们回到 Docker 主机来创建一个容器。你可以使用已有的 Docker 主机或者使用为本书准备的 Vagrant 镜像，如下所示。

```
$ git clone https://github.com/how2dock/docbook
$ cd ch03/simple
$ vagrant up
$ vagrant ssh
```

图 3-1 显示了这个 Vagrant 镜像的网络配置情况。它只有一个 NAT 网络接口用来访问外部

网络。这台主机内部有一个 Linux 网桥 docker0，此外还有两个容器。

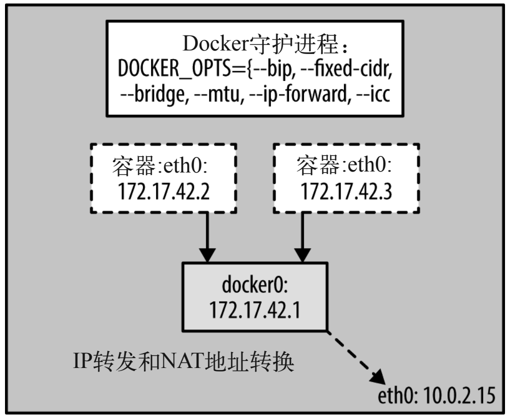


图 3-1：单台 Docker 主机网络拓扑图

连接到这台主机之后，如果查看一下网络接口设备列表，将会看到有环回设备、一个 eth0 设备和 docker0 网桥设备，如图中显示的那样。

当虚拟机启动时，Docker 也会随之启动，并自动创建一个网桥，然后为其分配一个子网，如下所示。

```
$ ip -d link show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN ...
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00 promiscuity 0
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast ...
    link/ether 08:00:27:98:a7:ad brd ff:ff:ff:ff:ff:ff promiscuity 0
3: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue ...
    link/ether 00:00:00:00:00:00 brd ff:ff:ff:ff:ff:ff promiscuity 0
    bridge
```

现在让我们启动一个容器并查看一下它的网络接口信息，如下所示。

```
$ docker run -ti --rm ubuntu:14.04 bash
root@4e3ffb9bc381:/# ip addr show eth0
6: eth0: <BROADCAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:ac:11:2a:03 brd ff:ff:ff:ff:ff:ff
    inet 172.17.42.3/24 scope global eth0
...
```

实际上，这个容器拥有一个在 172.17.42.1/16 网段的 IP 地址（即 172.17.42.3），以及一个虚拟网络接口（即下面将要看到的 veth450b81a 设备），这个网络接口是 Docker 创建容器时自动创建并连接到桥接设备的。

可以通过在 Docker 主机上执行 ip（或 ifconfig）命令来查看这个设备。如果安装了 bridge-utils 包，也可以使用 brctl 工具。ip 命令提供了一组用于在 Linux 中控制 TCP/IP 通信的工具。可以在 Linux 基金会主页（https://wiki.linuxfoundation.org/networking/iproute2）上看到该项目的文档。

```
$ ip -d link show
...
3: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP ...
    link/ether aa:85:e0:61:69:2d brd ff:ff:ff:ff:ff:ff promiscuity 0
    bridge
7: veth450b81a: <BROADCAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast master docker0
    link/ether aa:85:e0:61:69:2d brd ff:ff:ff:ff:ff:ff promiscuity 1
    veth
$ brctl show
bridge name    bridge id                STP enabled     interfaces
docker0        8000.aa85e061692d        no              veth450b81a
```

可以在容器中 ping 一下网关 172.17.42.1（即 `docker0`）、位于同一 Docker 主机上的其他容器和外部网络。

你可以从其他控制台启动一个新容器来互相 ping 一下。确认一下第二个容器的网络接口也连接到了网桥设备。由于没有任何丢弃数据包的 iptables 规则，因此这两个容器可以通过任何端口互相通信。

3.4.3　讨论

访问外部的网络流量将会通过 IP 转发功能转发到 Docker 主机的其他网络接口上，并通过 iptables 的地址伪装规则进行 NAT 地址转换。你可以在 Docker 主机上通过下面的命令来检查一下 IP 转发功能是否已经启用。

```
$ cat /proc/sys/net/ipv4/ip_forward
1
```

尝试一下关闭 IP 转发功能，你就会发现容器将不能再连接到外部网络，如下所示。

```
# echo 0 > /proc/sys/net/ipv4/ip_forward
```

也可以查看为外部通信而创建的用于 IP 伪装的 NAT 规则，如下所示。

```
$ sudo iptables -t nat -L
...
Chain POSTROUTING (policy ACCEPT)
target     prot opt source               destination
MASQUERADE  all  --  172.17.42.0/24       anywhere
...
```

在范例 3.7 中，你将会看到如何从零开始手动对 Docker 容器进行设置。

3.4.4　参考

- Docker 网络官方文档（https://docs.docker.com/articles/networking/）

3.5 选择容器网络模式

3.5.1 问题

在启动一个容器时，你希望能够选择一个特定的网络命名空间。对某些在容器中运行的特定应用程序，你可能希望使用一个与默认网桥网络不一样的网络设置，或者你根本就不需要任何网络功能。

3.5.2 解决方案

在范例 3.4 中，我们使用默认的 `docker run` 命令启动了一个容器。这将会把新启动的容器绑定到一个 Linux 网桥设备，并为容器创建相应的网络接口设备。借助于 Linux 的 IP 转发功能和 Docker 引擎所管理的 iptables 规则，Docker 为容器提供了对外部网络和 NAT 功能的访问。

但是，我们也可以以一种不同的网络模式来启动新容器。通过使用 `docker run` 命令的 `--net` 选项，我们可以选择主机模式、无网络模式，或者选择与其他容器共享网络的模式。

让我们在 Docker 主机上，通过使用 `--net=none` 选项来启动一个不带任何网络功能的容器，如下所示。

```
$ docker run -it --rm --net=none ubuntu:14.04 bash
root@3a22f5076f9a:/# ip -d link show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00 promiscuity 0
root@3a22f5076f9a:/# route
        Kernel IP routing table
        Destination     Gateway     Genmask     Flags Metric Ref    Use Iface
```

在查看网络设备时，你会发现这个容器只有一个本地环回设备，没有其他网络设备，也没有路由信息。如果需要使用网络，只能手动进行设置（参见范例 3.7）。

现在让我们通过 `--net=host` 选项来启动一个 host 模式的容器，如下所示。

```
$ docker run -it --rm --net=host ubuntu:14.04 bash
root@foobar-server:/# ip -d link show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00 promiscuity 0
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state ...
    link/ether 08:00:27:98:a7:ad brd ff:ff:ff:ff:ff:ff promiscuity 0
3: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state ...
    link/ether c6:4b:6b:b7:4b:98 brd ff:ff:ff:ff:ff:ff promiscuity 0
    bridge
```

查看这个容器中的网络设备，将会看到与宿主机完全相同的结果，包括 `docker0` 网桥设备。这也意味着容器中的进程被隔离在它自己的命名空间中，它的资源被 cgroups 限制，容器的网络命名空间和宿主是相同的。你也看到了，在上面的例子中，容器中的主机名与宿主机的主机名也是一样的（可以在使用 host 网络模式启动容器时指定 `h` 选项来为容器设置主机名）。注意，如果这样，你将不能在容器中对网络重新进行任何修改。比如，你不能将

一个网络设备关掉，如下所示。

```
root@foobar-server:/# ifconfig eth0 down
SIOCSIFFLAGS: Operation not permitted
```

虽然 host 网络模式很方便，但是在使用时有很多事情需要特别注意。

> 有时候通过 `--net=host` 选项启动一个容器可能会比较危险，尤其是同时指定 `--privileged=true` 参数启动一个特权容器的时候。host 网络模式可能会导致你在容器中不小心对主机中的网络进行了修改。如果你打算以 root 权限在一个特权容器中运行一个程序，并在启动时指定 `--net=host` 选项，那么应用程序的漏洞就可能会导致入侵者控制你的整个 Docker 主机的网络。尽管如此，host 网络模式非常适合那些对网络 I/O 性能特别敏感的进程。

最后一种可以选择的容器网络模式是与已经启动的容器共享一个网络命名空间。让我们启动一个容器并将其主机名设置为 cookbook，如下所示。

```
$ docker run -it --rm -h cookbook ubuntu:14.04 bash
root@cookbook:/# ifconfig
eth0      Link encap:Ethernet  HWaddr 02:42:ac:11:00:02
          inet addr:172.17.0.2  Bcast:0.0.0.0  Mask:255.255.0.0
                ...
```

从控制台上可以看到容器的主机名被设置为 cookbook，IP 被设置为 172.17.0.2，这个容器被连接到了 `docker0` 网桥设备上。

接着让我们再启动第二个容器，该容器将使用第一个容器的网络命名空间。首先列出正在运行中的容器，找到刚才启动的容器名。启动第二个容器时需要使用 `--net=container:CONTAINER_NAME_OR_ID` 这样的方式，如下所示。

```
$ docker ps
CONTAINER ID    IMAGE         COMMAND  ...   NAMES
cc7f72826c36    ubuntu:14.04  "bash"   ...   cocky_galileo
$ docker run -ti --rm --net=container:cocky_galileo ubuntu:14.04 bash
root@cookbook:/# ifconfig
eth0      Link encap:Ethernet  HWaddr 02:42:ac:11:00:02
          inet addr:172.17.0.2  Bcast:0.0.0.0  Mask:255.255.0.0
                ...
```

如上面的输出所示，新的容器具有与第一个启动的容器相同的主机名，当然也具有相同的 IP。在每个容器中的进程将是隔离的，存在于自己的进程命名空间中，但是它们共享同一网络命名空间，并可以通过环回设备进行通信。

3.5.3 讨论

使用哪种网络模式取决于你要运行的应用程序，以及你希望采用什么样的网络结构。Docker 的网络模型非常灵活，可以让你在容器进程之间构建任何网络拓扑结构和安全网络方案。

3.5.4　参考

- 理解 Docker 容器网络（https://docs.docker.com/articles/networking/#container-networking）

3.6　配置Docker守护进程iptables和IP转发设置

3.6.1　问题

也许你不喜欢 Docker 默认会启用 IP 转发功能并且修改你的 iptables 规则表。你希望能对 Docker 主机上容器之间以及容器与外部网络之间的网络流量进行更精细的控制。

3.6.2　解决方案

Docker 默认的网络设置对很多人来说也许已经足够了。但是，你可以在启动 Docker 守护进程时通过 `--ip-forward=false` 和 `--iptables=false` 参数对 Docker 的网络进行定制。本范例将会为你介绍如何进行这样的定制。

要想修改 Docker 的默认网络设置，首先需要停止 Docker 守护进程。在类似 Ubuntu 这样基于 Debian 的发行版中，编辑 /etc/default/docker 文件，然后将这两个参数设为 `false`（在 CentOS/RHEL 系统下编辑 /etc/sysconfig/docker 文件），如下所示。

```
$ sudo service docker stop
$ sudo su
# echo DOCKER_OPTS=\"--iptables=false --ip-forward=false\" >> /etc/default/docker
# service docker restart
```

必须在重启 Docker 守护进程之前，手动删除 postrouting 规则然后将 IP 转发标志设置为 0。在 Docker 主机上可以运行下面的命令来完成上述内容。

```
# iptables -t nat -D POSTROUTING 1
# echo 0 > /proc/sys/net/ipv4/ip_forward
# service docker restart
```

在这种配置下，经过 Docker 网桥 `docker0` 的通信流量都不会被转发到容器的网络接口上，也不会添加 postrouting 和 masquerading 规则。这意味着所有来自容器对外部的网络访问请求都会被丢弃。

可以启动一个新容器来尝试访问一下外部网络，以验证上述结论，比如像下面这样。

```
$ docker run -it --rm ubuntu:14.04 bash
WARNING: IPv4 forwarding is disabled.

root@ba12d578e6c8:/# ping -c 2 -W 5 8.8.8.8
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.

--- 8.8.8.8 ping statistics ---
2 packets transmitted, 0 received, 100% packet loss, time 1009ms
```

要想手动恢复容器对外部网络的访问，需要在 Docker 主机上启用 IP 转发并设置

postrouting 规则，像下面这样。

```
# echo 1 > /proc/sys/net/ipv4/ip_forward
# iptables -t nat -A POSTROUTING -s 172.17.0.0/16 -j MASQUERADE
```

然后回到刚才容器的终端控制台，再试着 ping 一下 8.8.8.8，这时候通信流量应该能被路由到宿主机之外了。

> 如果启用了 Docker 守护进程的 --iptables=false 标志，就不能再对容器之间的通信进行限制（即使用 --icc=false），因此这时 Docker 已经不能对 iptables 规则进行管理了。这也意味着，所有连接到同一 Docker 网桥设备的容器都可以使用任何端口进行通信。可以阅读接下来的讨论部分来了解这方面的内容。

3.6.3 讨论

默认情况下，Docker 守护进程可以对宿主机上的 iptables 规则进行修改。也就是说，Docker 可以添加用于限制容器之间网络通信的 iptables 规则，以实现容器之间的网络隔离。

如果禁止了 Docker 对 iptables 规则的修改，它将不能添加用于限制容器间通信的规则。

如果允许 Docker 去修改 iptables 规则，你可以为 Docker 守护进程添加 --icc=false 选项。这将会为 Docker 网桥上的数据包添加一个默认丢弃的规则，容器之间也不能互相访问。

可以修改 Docker 配置文件（在 Ubuntu/Debian 系统下为 /etc/default/docker，在 CentOS/RHEL 下为 /etc/sysconfig/docker），添加 --icc=false 选项。重启 Docker 守护进程后再启动两个容器，你就会看到这两个容器之间互相是 ping 不通的。

鉴于这大大限制了容器间互相通信的能力，要如何才能让两个容器互相通信呢？可以通过容器链接来解决，这会创建一个专用的 iptables 规则（参见范例 3.3）。

允许从 Docker 主机 ping 所有容器，如下所示。

```
$ sudo iptables -A DOCKER -p icmp --icmp-type echo-request -j ACCEPT
$ sudo iptables -A DOCKER -p icmp --icmp-type echo-reply -j ACCEPT
```

3.7 通过Pipework理解容器网络

3.7.1 问题

Docker 内建的网络功能已经很不错了，但是你想使用一种实用的方法，利用传统的网络工具为容器创建网络接口。

3.7.2 解决方案

这是一个比较高级的范例，意在提供关于 Docker 网络机制的更深入的知识。如果只是使用 Docker，本范例的内容和这里面提到的工具都不是必需的。但是，为了更好地理解 Docker

网络，你可能想去试用一下 Pipework（https://github.com/jpetazzo/pipework）。Pipework 是由 Docker 公司的 Jerome Petazzoni 在 2013 年创建的工具，它通过 cgroups 和网络命名空间来创建容器网络。最初它只支持纯 LXC 容器，不过现在已经支持 Docker 容器了。如果通过 `--net=none` 选项启动一个容器，那么你可以非常方便地使用 Pipework 对容器进行网络设置。如果你想获得关于 Docker 网络更详细的知识，这将是一个很好的锻炼机会，虽然我们在日常使用和生产环境下并不会这么用。

> 并不是只能使用 Pipework 来运行 Docker 或者管理容器的连通性。这个范例是为那些希望通过手动在容器网络命名空间创建网络协议栈，并想获得更高级知识的读者准备的。你可以使用 Pipework，阅读一下它的 bash 脚本，你将会学到更多的知识，你将学习到创建一个容器网络所需的所有详细到每一步的指令。

虽然通过 Pipework 能实现的功能几乎都内置在 Docker 中了，但是 Pipework 仍不失为一个伟大的工具：使用该工具可以对 Docker 网络进行逆向工程，深入了解容器之间以及容器与外部之间是如何通信的。本范例将介绍几个例子，你可以解构 Docker 的网络功能，更得心应手地对不同的网络命名空间进行控制。

Pipework 是一个单一的 bash 脚本文件，你可以免费下载（https://raw.githubusercontent.com/jpetazzo/pipework/master/pipework）。为了方便使用，我创建了一个 Vagrant 虚拟机，并将 Pipework 放到了里面。你可以克隆这个 Git 仓库，然后启动 Vagrant 虚拟机以使用 Pipework。

```
$ git clone https://github.com/how2dock/docbook
$ cd ch03/simple
$ vagrant up
$ vagrant ssh
vagrant@foobar-server:~$ cd /vagrant
vagrant@foobar-server:/vagrant$ ls
pipework  Vagrantfile
```

让我们通过 `--net=none` 选项启动一个不带网络功能的容器，如同范例 3.5 那样。

```
$ docker run -it --rm --net none --name cookbook ubuntu:14.04 bash
root@556d04d8637e:/# ip -d link show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode ...
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00 promiscuity 0
```

在该 Docker 主机的另一个终端中，让我们使用 Pipework 来创建一个网桥设备 `br0`，为容器分配一个 IP 地址，然后设置从容器到网桥的正确路由信息，如下所示。

```
$ cd /vagrant
$ sudo ./pipework br0 cookbook 192.168.1.10/24@192.168.1.254
Warning: arping not found; interface may not be immediately reachable
```

在容器中，确认 `eth1` 设备已启用，并且路由信息已经设置好，如下所示。

```
root@556d04d8637e:/# ip -d link show eth1
7: eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP ...
    link/ether a6:95:12:b9:8f:55 brd ff:ff:ff:ff:ff:ff promiscuity 0
```

```
    veth
root@556d04d8637e:/# route
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
default         192.168.1.254   0.0.0.0         UG    0      0        0 eth1
192.168.1.0     *               255.255.255.0   U     0      0        0 eth1
```

现在，如果查看该 Docker 主机上的网络设备列表，除了默认的 `docker0` 网桥设备，你还会看到多出来一个 `br0` 设备；如果查看网桥信息（使用 bridge-utils 软件包的 `brctl` 命令），你将会看到由 `pipework` 创建并连接到 `br0` 的虚拟以太网接口，如下所示。

```
$ ip -d link show
...
3: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state \
DOWN mode DEFAULT group default
    link/ether 00:00:00:00:00:00 brd ff:ff:ff:ff:ff:ff promiscuity 0
    bridge
8: br0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state \
DOWN mode DEFAULT group default
    link/ether 22:43:24:f5:91:7e brd ff:ff:ff:ff:ff:ff promiscuity 0
    bridge
10: veth1pl31668: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc \
pfifo_fast master br0 state DOWN mode DEFAULT group default qlen 1000
    link/ether 22:43:24:f5:91:7e brd ff:ff:ff:ff:ff:ff promiscuity 1
    veth
$ brctl show
bridge name     bridge id               STP enabled     interfaces
br0             8000.224324f5917e       no              veth1pl31668
docker0         8000.000000000000       no
```

到这里，你就已经可以在宿主机上访问容器了，也可以从容器 `cookbook` 中访问其他容器了。但是，如果你尝试访问宿主机外部，就会发现不能正常工作。这是因为没有相应的 NAT 地址伪装规则，而这个 NAT 地址伪装规则在默认情况下会由 Docker 自动添加。在 Docker 宿主机上添加相应的 iptables 规则，然后尝试在容器交互终端上 ping `8.8.8.8`，如下所示。

```
# iptables -t nat -A POSTROUTING -s 192.168.0.0/16 -j MASQUERADE
```

在容器中，确认是否可以访问到 Docker 主机的外部，如下所示。

```
root@556d04d8637e:/# ping 8.8.8.8
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=61 time=22.6 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=61 time=23.8 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=61 time=23.9 ms
```

Pipework 还可以做很多事情，所以请先浏览一下它的 README 文件（https://github.com/jpetazzo/pipework），并阅读一下它的 bash 脚本文件，以获得对网络命名空间更深入的理解。

3.7.3 讨论

尽管 Pipework 非常强大，你可以在启动容器时使用 `--net=none` 选项，然后通过 Pipework

创建正常工作的网络协议栈，但是 Pipework 还隐藏了一些控制容器网络命名空间的细节。如果阅读一下 Pipework 的代码，你就会知道它是如何工作的。Docker 文档（https://docs.docker.com/articles/networking/#container-networking）对此也有详细的说明。不管是对于网络还是对于容器，这都是一个很好的练习，推荐各位阅读一下这些文档。

> 这部分的讨论不是专门来讲 Pipework 的，而是为了让你了解构建容器网络协议栈所需要的步骤。这对充分理解容器网络和对 Docker 的工作原理进行逆向工程非常有用。

让我们再来看一条 Pipework 命令，如下所示。

```
$ sudo ./pipework br0 cookbook 192.168.1.10/24@192.168.1.254
```

这条命令完成了下面这些工作。

- 在宿主机上创建网桥设备 `br0`。
- 分配 IP 地址 `192.168.1.254`。
- 在容器内部创建一个网络接口，并为其分配 IP 地址 `192.168.1.10`。
- 最后在容器内部添加路由信息，将网桥设置为默认网关。

下面我们不使用 Pipework，一步一步地进行上述设置。首先我们添加一个 `br0` 的网桥设备，并为其分配 IP 地址 `192.168.1.254`。如果你之前使用过虚拟机进行虚拟化工作，应该会对这些操作比较熟悉。如果你还不太熟悉，那么请跟着操作：使用 `brctl` 工具创建一个网桥设备，用 `ip` 名为网桥添加 IP 地址，最后启用网桥设备。

如果你已经按照前面的步骤进行了操作，你可能需要先删除已经存在的 `br0` 网桥，如下所示。

```
$ sudo ip link set br0 down
$ sudo brctl delbr br0
```

然后就可以再做一遍之前的配置了，不过这次都是手动进行的。

```
$ sudo brctl addbr br0
$ sudo ip addr add 192.168.1.254/24 dev br0
$ sudo ip link set dev br0 up
```

与完全网络虚拟化相比，这其中巧妙的部分是你的操作对象是容器，并且它的网络协议栈是宿主机上一个不同的网络命名空间。要想为容器分配网络接口，你需要把这个网络接口分配给容器将使用的网络命名空间。分配到某个特定网络命名空间的接口是一个虚拟以太网接口对。这些设备对就像是一个管道：管道的一端位于容器的网络命名空间中，另一端位于在 Docker 宿主机上所创建的网桥设备上。

下面，就让我们来创建一个 `veth` 对 foo、bar，并将 foo 连接到网桥 `br0` 上，如下所示。

```
$ sudo ip link add foo type veth peer name bar
$ sudo brctl addif br0 foo
$ sudo ip link set foo up
```

可以通过 ip -d link show 命令来检查一下上面操作的结果。该命令会创建一个新的网桥设备 br0，并且将一个名为 foo 的 veth 设备连接到这个网桥上，如下所示。

```
$ ip -d link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT \
group default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00 promiscuity 0
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state \
UNKNOWN mode DEFAULT group default qlen 1000
    link/ether 08:00:27:98:a7:ad brd ff:ff:ff:ff:ff:ff promiscuity 0
3: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state \
DOWN mode DEFAULT group default
    link/ether 00:00:00:00:00:00 brd ff:ff:ff:ff:ff:ff promiscuity 0
    bridge
6: br0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP mode \
DEFAULT group default
    link/ether ee:7d:7e:f7:6f:18 brd ff:ff:ff:ff:ff:ff promiscuity 0
    bridge
8: foo: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast master br0 \
state UP mode DEFAULT group default qlen 1000
    link/ether ee:7d:7e:f7:6f:18 brd ff:ff:ff:ff:ff:ff promiscuity 1
    veth
$ brctl show
bridge name     bridge id               STP enabled     interfaces
br0             8000.ee7d7ef76f18       no              foo
docker0         8000.000000000000       not
```

请不要将你的 veth 对命名为常见的 eth0 或 eth1，因为这会与宿主机上已有的硬件设备冲突。

让我们更深入地了解一下，当你以 --net= none 的方式启动容器时，它也会创建相应的网络命名空间，但是其中除了环回设备之外，没有其他网络设备。现在，你想要对容器的网络进行配置（例如，增加一个网络接口，设置路由信息）。首先，你需要找到网络命名空间 ID。Docker 将容器的网络命名空间保存在 /var/run/docker/netns 下面，这并不是 Linux 系统默认的命名空间保存位置。为了能够正常地使用 ip 工具，你需要使用一些非常规的技巧，并将 /var/run/docker/netns 链接到 /var/run/netns，后者也是 ip 命令默认查找网络命名空间的位置。完成了上述操作后，你就可以列出现有的网络命名空间。然后你会发现，该容器网络命名空间的 ID 就是是容器的 ID。

```
$ cd /var/run
$ sudo ln -s /var/run/docker/netns netns
$ sudo ip netns
c785553b22a1
$ NID=$(sudo ip netns)
```

接着，我们就通过 ip link set netns 命令将 veth 的另一端 bar 放入到容器的命名空间中，并通过 ip netns exec 命令在该命名空间下为其设置一个名称和 MAC 地址，如下所示。

```
$ sudo ip link set bar netns $NID
$ sudo ip netns exec $NID ip link set dev bar name eth1
$ sudo ip netns exec $NID ip link set eth1 address 12:34:56:78:9a:bc
$ sudo ip netns exec $NID ip link set eth1 up
```

最后一步工作是为容器内的 eth1 设备分配一个 IP 地址，并设置一个默认的路由。这样，该容器就可以通过网络连接到 Docker 宿主机以及外部网络了，如下所示。

```
$ sudo ip netns exec $NID ip addr add 192.168.1.1/24 dev eth1
$ sudo ip netns exec $NID ip route add default via 192.168.1.254
```

这就是需要做的全部工作。目前为止，该容器已经与上面使用 Pipework 通过一条命令创建的容器具有相同的网络功能。

记住，如果你想访问容器外部网络，就要添加如下的 IP NAT 地址伪装规则。

```
$ sudo iptables -t nat -A POSTROUTING -s 192.168.0.0/24
-j MASQUERADE
```

3.7.4　参考

- Pipework 关于各种使用场景的 README 文件（https://github.com/jpetazzo/pipework）
- Docker 容器网络原理（https://docs.docker.com/articles/networking/#container-networking）
- `ip netns` 手册（http://man7.org/linux/man-pages/man8/ip-netns.8.html）
- Linux 网络命名空间简介（http://blog.scottlowe.org/2013/09/04/introducing-linux-network-namespaces/）

3.8　定制Docker网桥设备

3.8.1　问题

你想创建一个自己的 Docker 网桥设备，以取代 Docker 默认的网桥。

3.8.2　解决方案

创建一个网桥设备，并在启动 Docker 守护进程时使用这个网桥设备。

在范例 3.7 的解决方案中，你已经学习了如何在以 `--net=none` 选项启动的容器中构建完整的网络协议栈。范例 3.7 也介绍了如何创建一个网桥设备。这里我们将重用在范例 3.7 中讨论过的内容。

首先，让我们来停止 Docker 守护进程，删除 Docker 默认创建的 `docker0` 网桥设备，然后创建一个新的名为 `cookbook` 的网桥设备，如下所示。

```
$ sudo service docker stop
$ sudo ip link set docker0 down
$ sudo brctl delbr docker0
```

```
$ sudo brctl addbr cookbook
$ sudo ip link set cookbook up
$ sudo ip addr add 10.0.0.1/24 dev cookbook
```

现在，新的网桥设备已经启用，你可以编辑 Docker 守护进程的配置文件，然后重启 Docker 守护进程（比如在 Ubuntu 上），如下所示。

```
$ sudo su
# echo 'DOCKER_OPTS="-b=cookbook"' >> /etc/default/docker
# service docker restart
```

你可以启动一个容器并查看 Docker 为它分配的 IP 地址，然后测试网络连通性，如下所示。

```
root@c557cdb072ba:/# ip addr show eth0
10: eth0: <BROADCAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:0a:00:00:02 brd ff:ff:ff:ff:ff:ff
    inet 10.0.0.2/24 scope global eth0
        ...
```

像我们预测的那样，Docker 也为新的网桥设备自动创建了 NAT 规则，如下所示。

```
$ sudo iptables -t nat -L
...
Chain POSTROUTING (policy ACCEPT)
target     prot opt source               destination
MASQUERADE  all  --  10.0.0.0/24          anywhere
```

3.8.3 讨论

尽管你可以手动完成以上操作，但是上面创建的 `cookbook` 网桥与默认的 `docker0` 网桥设备没有什么差别。

如果想修改 Docker 容器通过默认的网络模式（比如网桥）启动时所分配的 IP 地址范围，可以使用 `--bip` 选项。你也可以使用 `--fixed-cidr` 选项来限制这个 IP 范围，或者使用 `--mtu` 选项来设置 MTU 的大小。

要想关闭一个网桥设备，可以使用以下命令。

```
$ sudo ip link set cookbook down
$ sudo brctl delbr cookbook
```

3.9 在Docker中使用OVS

3.9.1 问题

你已经知道了如何使用自定义的网桥设备来为你的 Docker 容器设置网络（参见范例 3.8），但是你想使用虚拟交换机软件 Open vSwitch（OVS）来代替标准的 Linux 网桥设备。也许你想构建自己的 GRE 或基于 VXLAN 的覆盖网络，或者你想使用网络控制器来构建一个软件定义网络解决方案。通过使用 OpenFlow 协议（https://www.opennetworking.org/openflow）和 OVSDB 管理协议（https://tools.ietf.org/html/rfc7047），OVS 提供了非常实用

的扩展和控制功能。

3.9.2 解决方案

在 Docker 1.7 中，还没有对 Open vSwitch 的原生支持。虽然也可以使用 Open vSwitch，但是你可能需要类似 Pipework（参见范例 3.7）的工具，或者一个称为 ovs-docker（https://github.com/openvswitch/ovs/blob/master/utilities/ovs-docker）的软件，或者手动构建容器的网络协议栈。在将来的 DockerNetwork 版本中（参见范例 3.14），应该会有对 Open vSwitch 的原生支持。

使用 Open vSwitch（http://openvswitch.org）作为网桥，并在 Docker 守护进程的配置文件中指定该网桥设备。

首先需要在 Docker 主机上安装 OVS 的软件包，比如，在 Ubuntu 14.04 上使用如下命令。

```
$ sudo apt-get -y install openvswitch-switch
```

如果你想使用最新版本的 Open vSwitch，可以从它的源代码（https://github.com/openvswitch/ovs）进行构建，这也非常简单。

现在创建一个桥接设备并启动它，如下所示。

```
$ sudo ovs-vsctl add-br ovs-cookbook
$ sudo ip link set ovs-cookbook up
```

现在你就可以使用 Pipework（参见范例 3.7）来构建连接到该 Open vSwitch 上的容器的网络协议栈了。

在启动容器时，不指定任何网络协议栈（即 `--net=none`），如下所示。

```
$ docker run -it --rm --name foobar --net=none ubuntu:14.04 bash
root@8fda6e33eb88:/#
```

在该 Docker 主机的另一个终端中，使用 Pipework 在你的 foobar 容器中创建一个网络接口（需要确保已经安装了 Pipework），如下所示。

```
$ sudo su
# ./pipework ovs-cookbook foobar 10.0.0.10/24@10.0.0.1
# ovs-vsctl list-ports ovs-cookbook
veth1pl31350
```

你的网桥设备也会由 Pipework 设置一个 `10.0.0.1` 的 IP 地址，如下所示。

```
$ ifconfig
ovs-cookbook  Link encap:Ethernet  HWaddr 36:b1:d3:e5:fc:44
          inet addr:10.0.0.1  Bcast:0.0.0.0  Mask:255.255.255.0
                  ...
```

该容器现在也拥有了一个网络设备接口，如下所示。

```
root@8fda6e33eb88:/# ifconfig
eth1      Link encap:Ethernet  HWaddr 52:fe:9f:78:b7:fc
          inet addr:10.0.0.10  Bcast:0.0.0.0  Mask:255.255.255.04
....
```

你也可以通过使用 `ip netns` 命令来手动创建这个网络接口，正如范例 3.7 的讨论部分所述的那样。

3.9.3 参考

- Open vSwitch 官方网站（http://openvswitch.org）

3.10 在Docker主机间创建GRE隧道

3.10.1 问题

你需要在位于不同 Docker 主机上的容器中使用它们独立的 IP 地址进行网络通信。

3.10.2 解决方案

有几个范例（参见范例 3.11 和范例 3.13）介绍了一些可以用于生产环境的解决方案。在本范例的例子中，我们将会介绍如何一步一步地构建多主机网络环境，为帮助你理解 Docker 网络奠定一个坚实的基础。

为了在多主机环境下为容器提供网络互连功能，可以构建一个通用路由封装（Generic Routing Encapsulation，GRE）来对 IPv4 通信进行封装，并为容器之间互连提供基于容器私有地址的路由。为了展示这一技术，你需要启动两台 Docker 主机并进行网络配置，如图 3-2 所示。

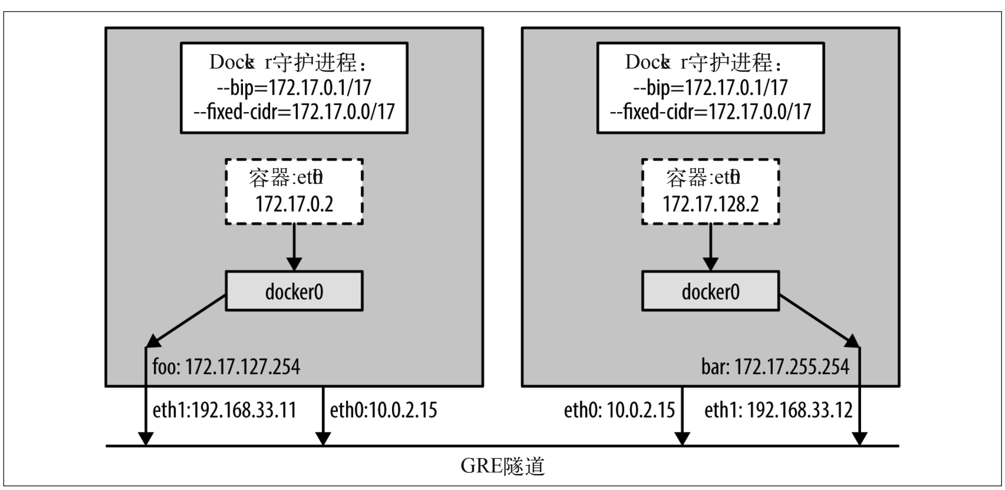


图 3-2：两台主机 GRE 隧道覆盖网络图

host-1 的 IP 地址为 192.168.33.11。我们将会为网桥 docker0 分配 IP 地址 172.17.0.1，并创建一个 GRE 隧道端点，其 IP 地址为 172.17.0.2。Docker 会为在该主机上启动的容器分配一个位于 172.17.0.0/17 网络中的 IP 地址。

host-2 的 IP 地址为 192.168.33.12。我们将会为网桥 docker0 分配 IP 地址 172.17.128.1，并创建一个 GRE 隧道端点，其 IP 地址为 172.17.128.2。Docker 会为在该主机上启动的容器分配一个位于 172.17.128.0/17 网络中的 IP 地址。

将一个 /16 位的网络分割为两个 /17 位的网络，并将这两个子网分配给不同的主机，这样就能确保两台主机上的容器不会出现 IP 冲突的问题。

你可以使用这个 Vagrantfile（https://github.com/how2dock/docbook/blob/master/ch03/gresimple/Vagrantfile）的配置来运行该示例。每台主机都安装了最新稳定版的 Docker 程序，以及两个网络设备接口：一个 NAT 接口用来提供对外部网络的连通性，一个私有网络的网络接口。

为了避免可能出现的错误，第一件需要做的事情是停止 Docker 引擎并且删除在配置 Docker 时创建的 docker0 网桥设备。需要在你所有的 Docker 主机上执行以下操作。

```
$ sudo su
# service docker stop
# ip link set docker0 down
# ip link del docker0
```

现在就可以在两台主机之间创建 GRE 隧道了。创建 GRE 隧道并不需要 Open vSwitch，只需要使用 ip 命令即可。如果使用了前面提到的 Vagrantfile 文件，请在第一台 IP 地址为 192.168.33.11 的主机上执行以下操作。

```
# ip tunnel add foo mode gre local 192.168.33.11 remote 192.168.33.12
# ip link set foo up
# ip addr add 172.17.127.254 dev foo
# ip route add 172.17.128.0/17 dev foo
```

如果没有使用前面提到的 Vagrantfile，那么你需要在最初的 ip tunnel 命令中，将指向 local 和 remote 端点的 IP 地址替换为你的两台 Docker 主机的实际 IP 地址。前面的四条命令首先创建了一个名为 foo 的 GRE 隧道。然后你启用了该隧道并为其分配了一个 IP 地址。接着你添加了一条路由信息，让所有发送到 172.17.128.0/17 的网络流量都经过该隧道。

在第二台主机上，执行前面的步骤来创建隧道的另一端。将隧道的另一端命名为 bar，然后添加路由信息让所有发送到 172.17.0.0/17 的流量都经过这个隧道，如下所示。

```
# ip tunnel add bar mode gre local 192.168.33.12 remote 192.168.33.11
# ip link set bar up
# ip addr add 172.17.255.254 dev bar
# ip route add 172.17.0.0/17 dev bar
```

一旦隧道启动，你可以验证一下是否能强制使用隧道并且能 ping 通。现在，就让我们在两台主机上启动 Docker。首先，需要对每个 Dockr 守护进程进行设置，指定分配给容器的正确子网，以及为 docker0 网桥指定正确的 IP 地址。要做到这一点，可以编辑 Docker 守护进程的配置文件，并使用 --bip 和 --fixed-cidr 这两个参数。

在 host-1 上，相应的配置项如下所示。

```
# echo DOCKER_OPTS=\"--bip=172.17.0.1/17 --fixed-cidr=172.17.0.0/17\" \
>> /etc/default/docker
```

host-2 上的配置项如下所示。

```
# echo DOCKER_OPTS=\"--bip=172.17.128.1/17 --fixed-cidr=172.17.128.0/17\" \
>> /etc/default/docker
```

如果你选择了不同的分区方案或者拥有超过两台的主机，那么请相应地重复上述操作。

> 由于 Docker 将会打开 IP 转发功能，所有经过 `docker0` 的通信流量将会被转发到 `foo` 和 `bar`，因此我们不需要将任何隧道的其中一端连接到网桥设备上。

现在，剩下的全部工作就是重启 Dcoker 了。然后你可以在每台主机上启动一个容器，你会看到，它们可以通过 Docker 分配的私有 IP 地址直接进行网络通信。

3.10.3　讨论

有多种方法可以构建一个 Docker 主机的覆盖网络。Docker Network（参见范例 3.14）将会在 Docker 1.8 中发布，允许你使用内建的功能来创建 VXLAN 覆盖网络。也有一些第三方的方案可供选择，比如 Weave（参见范例 3.11）或者 Flannel（参见范例 3.13）。随着 Docker 插件框架的日趋成熟，这些类型的功能也将会发生显著的变化。比如，Weave 和 Flannel 将会以 Docker 插件的形式提供，而不是作为独立的网络配置。

3.10.4　参考

- Vincent Viallet 发表于 Wiredcraft 的博客（http://wiredcraft.com/blog/multi-host-docker-network/），也是本范例灵感的来源。

3.11　在Weave网络上运行容器

——本范例由 Fintan Ryan 提供

3.11.1　问题

你想要为你在跨越多个数据中心的、规模从单台到数千台的主机上运行的容器创建一个网络。该网络具备自动 IP 地址分配功能，并且集成基于 DNS 的服务发现。

3.11.2　解决方案

使用来自 Weaveworks（http://weave.works）团队的 Weave（http://github.com/weaveworks/weave）。

为了帮助你更好地体验 Weave 网络，我创建了一个 Vagrantfile，它可以启动两台运行着

Ubuntu 14.04 的主机，并在其中安装 Docker、Weave 和另外两个示例容器。你可以像下面这样测试一下。

```
$ git clone https://github.com/how2dock/docbook.git
$ cd ch03/weavesimple
$ vagrant up
```

下面是我们在这个例子中使用到的 Vagrant 主机。

- `172.17.8.101 weave-gs-01`
- `172.17.8.102 weave-gs-02`

接着你需要在两台主机上启动 Weave 网络。需要注意的是，在第二台计算机上启动 Weave 网络时，你需要指定第一台主机的 IP 地址。

```
$ vagrant ssh weave-gs-01
$ weave launch
$ vagrant ssh weave-gs-02
$ weave launch 172.17.8.101
```

这时，你就创建了一个 Weave 网络，它会自动为新创建的容器分配 IP 地址，并集成基于 DNS 的服务发现。

新创建的容器会在 Weave 网络中运行，自动分配一个唯一的 IP 地址，并且通过 Docker 的 `-h` 选项将容器注册到 DNS。

接下来，你将在每台主机上启动容器。为了够轻松地使用 Weave 网络启动容器，你需要使用 `weave env` 命令来设置 `DOCKER_HOST` 环境变量，如下所示。

```
$ vagrant ssh weave-gs-01
$ eval $(weave env)
$ docker run -d -h lb.weave.local fintanr/myip-scratch
$ docker run -d -h hello.weave.local fintanr/weave-gs-simple-hw

$ vagrant ssh weave-gs-02
$ eval $(weave env)
$ docker run -d -h lb.weave.local fintanr/myip-scratch
$ docker run -d -h hello-host2.weave.local fintanr/weave-gs-simple-hw
```

上面的命令完成了两项工作。首先，在 Docker 主机上创建了一个简单的 Hello World 应用的容器。接着，通过 DNS 创建了一个容器的负载平衡服务，并将其命名为 `lb`。

让我们在该 Weave 网络中启动一个容器，并访问之前启动的另一个容器，如下所示。

```
$ vagrant ssh weave-gs-01
$ eval $(weave env)
$ C=$(docker run -d -ti fintanr/weave-gs-ubuntu-curl)
$ docker attach $C
root@ad6b7c0b1c6e:/#
root@ad6b7c0b1c6e:/# curl lb
Welcome to Weave, you probably want /myip
root@ad6b7c0b1c6e:/# curl lb/myip
10.128.0.2
root@ad6b7c0b1c6e:/# curl lb/myip
```

```
10.160.0.1
root@ad6b7c0b1c6e:/# curl hello
{
   "message" : "Hello World",
   "date” : "2015-07-09 15:59:50"
}
```

你也可以使用下面的脚本文件，来启动用于进行测试的容器。

```
$ ./launch-simple-demo.sh
```

3.11.3 讨论

使用 Weave 网络，你可以快速、轻松地在具有自动 IP 地址分配和服务发现功能的可扩展网络上启动容器。

在这个例子中，你在第一台主机 `weave-gs-01` 上创建了一个 Weave 路由容器。在第二台主机 `weave-gs-02` 上，你通过指定第一台主机的 IP 地址启动了另一个 Weave 路由容器。这条命令告诉位于 `weave-gs-02` 主机上的 Weave 要与位于 `weave-gs-01` 上的 Weave 一起工作。

在这之后，使用 Weave 启动的所有容器对于该 Weave 网络内的其他容器都是可见的，不管这个容器位于哪台主机上。每个容器都会自动分配一个同一网络内唯一的 IP 地址，如果通过 `-h` 来启动容器，该容器还会自动注册到 Weave 的 DNS 服务中。

为了确认已启动的容器，也可以使用 Weave `Scope`（参见范例 9.12）。在每台计算机上执行下面的命令。

```
$ scope launch
```

3.11.4 参考

- Weave 入门指南（http://weave.works/guides）

3.12 在AWS上运行Weave网络

——本范例由 Fintan Ryan 提供

3.12.1 问题

你希望在部署到 AWS 的主机实例中使用 Weave 网络和 Weave DNS。

3.12.2 解决方案

作为先决条件，你需要具备以下项目。

- 一个 AWS 账号
- 一组访问和安全 API 密钥
- 安装了 Ansible 和 boto 软件

为了方便在 AWS 上体验 Weave，我已经创建了一个可以在 EC2 上启动两台 Ubuntu 14.04 主机的 Ansible playbook，该 playbook 还会安装 Docker 和 Weave。我还提供了第二个 playbook，它会启动一个简单的应用程序，并采用 HAProxy 作为负载平衡服务，放置到这两台主机之前，如下所示。

```
$ git clone https://github.com/how2dock/docbook.git
$ cd ch03/weaveaws
$ ansible-playbook setup-weave-ubunu-aws.yml
```

你可以通过编辑 ansible_aws_variables.ym 文件来修改 AWS 区域和 AMI。

通过下面的命令来启动容器。

```
$ ansible-playbook launch-weave-haproxy-aws-demo.yml
```

我提供了一个脚本，它会连接到 HAProxy 容器，并快速对服务进行循环访问。每个容器都会返回一个 JSON 结果，并将它的主机名作为其 JSON 输出的一部分，如下所示。

```
$ ./access-aws-hosts.sh

Connecting to HAProxy with Weave on AWS demo

{
    "message" : "Hello Weave - HAProxy Example",
    "hostname" : ws1.weave.local",
    "date" : "2015-03-13 11:23:12"
}

{
    "message" : "Hello Weave - HAProxy Example",
    "hostname" : ws4.weave.local",
    "date" : "2015-03-13 11:23:12"
}

{
    "message" : "Hello Weave - HAProxy Example",
    "hostname" : ws5.weave.local",
    "date" : "2015-03-13 11:23:12"
}
....
```

3.12.3 讨论

在使用 Weave 网络时，你在应用前面放置了一个 HAProxy 作为负载平衡服务，负载平衡服务后面则是一些分布在多台主机上的运行着简单应用程序的容器。

3.12.4 参考

- Weave 入门指南（http://weave.works/guides）

3.13 在Docker主机上部署flannel覆盖网络

——本范例由 Eugene Yakubovich 提供

3.13.1 问题

你希望位于不同主机上的容器能不通过端口映射而直接通信。

3.13.2 解决方案

使用 `flannel` 为容器创建一个覆盖网络。每个容器都会分配一个能直接从其他主机访问到的 IP 地址。让我们从下面两个由 Vagrantfile 启动的虚拟机开始。

```
$ git clone https://github.com/how2dock/docbook.git
$ cd ch03/flannel
$ vagrant up
```

这个 Vagrantfile 定义了两个安装了 Docker、`etcd` 和 `flannel` 的虚拟机。master 节点上将会运行一个键值存储（`etcd`），`flannel` 依赖 `etcd` 进行协作。

然后，通过 `vagrant ssh master` 进入虚拟机启动 `etcd`，并让其在后台运行。

```
$ cd /opt/coreos/etcd-v2.0.13-linux-amd64
$ nohup ./etcd --listen-client-urls=http://0.0.0.0:2379 \
               --advertise-client-urls=http://192.168.33.10:2379 &
```

在启动 `flannel` 守护进程之前，需要先将覆盖网络的配置信息写入 `etcd`。请确保所选择的子网区间和已存在的 IP 地址不存在冲突，如下所示。

```
$ ./etcdctl set /coreos.com/network/config '{ "Network": "10.100.0.0/16" }'
```

现在，启动 `flannel` 守护进程。注意，`--iface` 选项指定的是 Vagrantfile 文件里设置的私有网络 IP 地址。`flannel` 将会在这个网络接口上转发封装好的数据包，如下所示。

```
$ cd /opt/coreos/flannel-0.5.1
$ sudo ./flanneld --iface=192.168.33.10 --ip-masq &
$ sudo ./mk-docker-opts.sh -c -d /etc/default/docker
```

`flannel` 将会租用一个 `/24` 子网并分配给 `docker0` 桥接设备。这个申请到的子网会被写入 /run/flannel/subnet.env 文件中。实用工具 `mk-docker-opts.sh` 则用来将这个文件转换为供 Docker 守护进程使用的一组命令行选项。

最后，启动 Docker 守护进程。你可以通过检查 `docker0` 桥接设备的 IP 地址来确认一切是否工作正常。它的地址应该在 `10.100.0.0/16` 子网范围内，如下所示。

```
$ sudo service docker start
$ ifconfig docker0
```

```
docker0   Link encap:Ethernet  HWaddr 56:84:7a:fe:97:99
          inet addr:10.100.63.1  Bcast:0.0.0.0  Mask:255.255.255.0
...
```

在 worker 节点上重复同样的步骤，启动 flannel 服务。由于 etcd 已经在 master 节点上启动了，所以在 worker 节点上不需要再启动了。相反，需要告诉 flannel 使用 master 节点上的 etcd 实例地址，如下所示。

```
$ cd /opt/coreos/flannel-0.5.1
$ sudo ./flanneld --etcd-endpoints=http://192.168.33.10:2379 \
                  --iface=192.168.33.11 --ip-masq &
$ sudo ./mk-docker-opts.sh -c -d /etc/default/docker
$ sudo service docker start
```

在两个虚拟机节点上都启动了 flannel 网络，分别在每个节点上运行一个简单的 busybox 容器。这两个容器都会具有一个可以从远程容器 ping 通的 IP 地址。

3.13.3　讨论

所有的 flannel 成员都通过 etcd 进行协调。在启动时，flannel 守护进程会从 etcd 读取覆盖网络的配置信息，以及在其他节点中正在被使用的所有子网信息。然后，它会挑选一个子网（默认情况下为 /24），并通过尝试在 etcd 中创建一个密钥来申请子网。如果这个密钥创建成功，就会获得该选定子网 24 小时的租约。其关联值则包含了该主机的 IP 地址。

接下来，flannel 会使用 TUN 设备创建 flannel0 网络接口。从 docker0 网桥路由到 flannel0 的 IP 分段将会被发送给 flannel 守护进程。它将 IP 分段封装在 UDP 数据包中，并根据 etcd 中保存的子网信息将 UDP 包转发给正确的主机。接收到该 UDP 数据包的 Docker 主机则从 UDP 包中对封装的 IP 数据进行解包，并通过 TUN 设备发送到 docker0。

flannel 持续监听 etcd 中 flannel 成员的变化，以保持其最新的状态。此外，守护进程会在租期到期前一小时进行续租操作。

3.14　在多台Docker主机中使用Docker Network

3.14.1　问题

尽管可以手动创建多主机之间的隧道网络（参见范例 3.10），但是你想充分利用新的 Docker Network 功能并使用 VXLAN 覆盖网络。

3.14.2　解决方案

Docker Network 是一个新的功能，目前在 Docker 实验版上可用。本范例只是让你提前体验一下该功能，将来的 Docker 发布版中将会正式包括该功能。



在本书编写之际，Docker Network 依赖 Consul 作为键值存储，使用 Serf 作为节点发现，使用标准的 Linux 网桥来构建 VXLAN 覆盖网络。由于 Docker Network 功能正在紧张开发之中，这些需求和方法可能会在不久的将来发生变化。

作为本书的惯例，我也准备了一个 Vagrantfile，这个 Vagrantfile 会启动三台虚拟机。一台虚拟机作为 Consul 服务器，其余两台则作为 Docker 主机。

实验版的 Docker 程序安装在两台 Docker 主机上，而运行着 Consul 的虚拟机上则安装了最新的稳定版 Docker。

具体配置如下所示。

- consul-server，Consul 服务节点，运行在 Ubuntu 14.04 上，IP 地址为 192.168.33.10。
- net-1，第一台 Docker 主机，运行在 Ubuntu 14.10 上，IP 地址为 192.168.33.11。
- net-2，第二台 Docker 主机，运行在 Ubuntu 14.10 上，IP 地址为 192.168.33.12。

图 3-3 显示了这一服务器的组成情况。

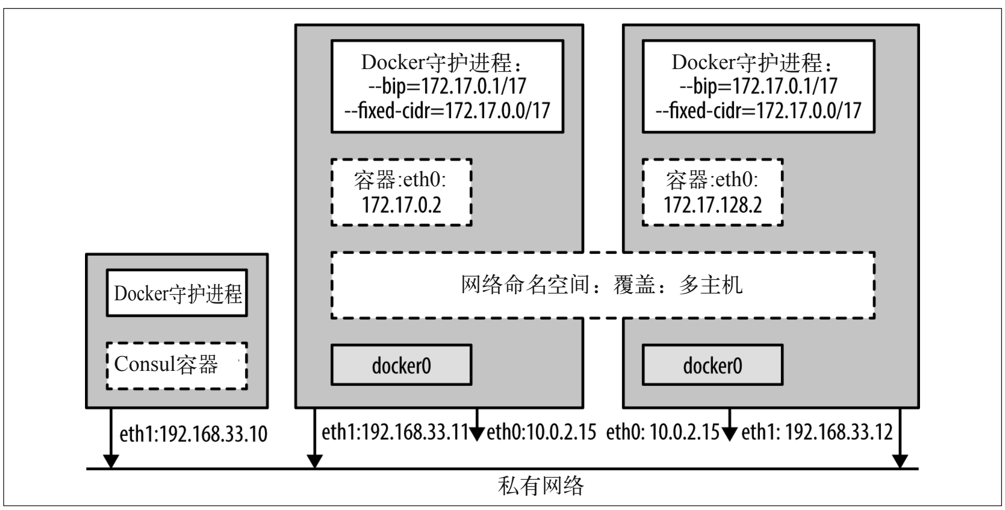


图 3-3：两台 Docker 主机组成的 VXLAN 覆盖网络和外加的 Consul 节点的网络拓扑图

克隆这个代码仓库，进入到 docbook/ch03/networks 目录，然后将工作交给 Vagrant。

```
$ git clone https://github.com/how2dock/docbook/
$ cd docbook/ch03/network
$ vagrant up
$ vagrant status
Current machine states:

consul-server             running (virtualbox)
net-1                     running (virtualbox)
net-2                     running (virtualbox)
```

你现在就可以 `ssh` 到 Docker 主机并启动容器。你会发现将要运行的是带 `-dev` 的实验版的 Docker 程序。实际上，根据所使用的 Docker 发布版本，你看到的版本号可能会与下面所示的略有不同。

```
$ vagrant ssh net-1
vagrant@net-1:~$ docker version
Client version: 1.9.0-dev
...<snip>...
```

通过列出可用的网络，可以验证 Docker Network 是否能正常工作，如下所示。

```
vagrant@net-1:~$ docker network ls
NETWORK ID          NAME                TYPE
4275f8b3a821        none                null
80eba28ed4a7        host                host
64322973b4aa        bridge              bridge
```

由于现在还没有发布任何服务，所以 `docker service ls` 将会返回一个空列表，如下所示。

```
$ docker service ls
SERVICE ID          NAME                NETWORK             CONTAINER
```

启动一个容器并检查容器中的 /etc/hosts 文件，如下所示。

```
$ docker run -it --rm ubuntu:14.04 bash
root@df479e660658:/# cat /etc/hosts
172.21.0.3      df479e660658
127.0.0.1       localhost
::1     localhost ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
172.21.0.3      distracted_bohr
172.21.0.3      distracted_bohr.multihost
```

在主机 net-1 上打开一个终端，再次查看一下 Docker 网络列表。你将会看到一个名为 `multihost` 的覆盖网络。

覆盖网络 `multihost` 是默认的网络。Vagrant 在初始化时对 Docker 守护进程进行了相应的设置。可通过查看 /etc/default/docker 文件的内容确认一下对 Docker 守护进程所做的设置。

```
vagrant@net-1:~$ docker network ls
NETWORK ID          NAME                TYPE
4275f8b3a821        none                null
80eba28ed4a7        host                host
64322973b4aa        bridge              bridge
b5c9f05f1f8f        multihost           overlay
```

现在，在另一个终端窗口中通过 `ssh` 连接到主机 net-2，检查一下它的网络和服务状态。你会看到 Docker 网络和 net-1 是一样的，默认的网络也是 `multihost`，网络类型也是 `overlay`。但是，在服务中会显示在主机 net-1 中启动的容器如下所示。

```
$ vagrant ssh net-2
vagrant@net-2:~$ docker service ls
SERVICE ID          NAME                NETWORK             CONTAINER
b00f2bfd81ac        distracted_bohr     multihost           df479e660658
```

在主机 net-2 上启动一个容器并查看 /etc/hosts 文件内容，如下所示。

```
vagrant@net-2:~$ docker run -ti --rm ubuntu:14.04 bash
root@2ac726b4ce60:/# cat /etc/hosts
172.21.0.4      2ac726b4ce60
127.0.0.1       localhost
::1     localhost ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
172.21.0.3      distracted_bohr
172.21.0.3      distracted_bohr.multihost
172.21.0.4      modest_curie
172.21.0.4      modest_curie.multihost
```

从上面的结果可以看到，除了刚才在主机 net-2 上启动的容器之外，还能看到在主机 net-1 上启动的容器。

当然，各个容器之间也可以互相 ping 通。

3.14.3 讨论

这种解决方案通过修改配置文件 /etc/default/docker，在启动时让 Docker 使用默认的覆盖网络。但是，你也可以使用非默认的覆盖网络。也就是说，你可以创建任意数量的覆盖网络，并且在不同网络中创建的容器是互相隔离的。

在前面的测试中，你在启动容器时使用了普通的 `-ti --rm` 参数。这样启动的容器将会被自动放置到默认的 Docker 网络中，默认的网络被设置为 `multihost`，类型为 `overlay` 网络。

但是你也可以创建自己的覆盖网络，并且在这个网络中创建新容器。让我们来看一下如何操作。首先，通过 `docker network create` 命令创建一个新的覆盖网络。

在 net-1 和 net-2 两台主机的其中一台上，执行下面的命令。

```
$ docker network create -d overlay foobar
8805e22ad6e29cd7abb95597c91420fdcac54f33fcdd6fbca6dd4ec9710dd6a4
$ docker network ls
NETWORK ID          NAME                TYPE
a77e16a1e394        host                host
684a4bb4c471        bridge              bridge
8805e22ad6e2        foobar              overlay
b5c9f05f1f8f        multihost           overlay
67d5a33a2e54        none                null
```

在第二台主机上也能自动看到这个新创建的网络。为了在新创建的网络中启动容器，你需要在执行 `docker run` 命令时使用 `--publish-service` 选项，如下所示。

```
$ docker run -it --rm --publish-service=bar.foobar.overlay ubuntu:14.04 bash
```

可以在启动容器时直接使用 `--publish-service` 选项指定一个新的覆盖网络，这个新的覆盖网络会自动创建。

现在再来查看一下 Docker 服务，如下所示。

```
$ docker service ls
SERVICE ID          NAME                NETWORK             CONTAINER
b1ffdbfb1ac6        bar                 foobar              6635a3822135
```

也可以在另一台 Docker 主机中重复上面这些步骤，在新的覆盖网络中启动另一个容器，然后检查 /etc/hosts 文件，并尝试在两个容器中互相 ping 对方。

3.15　深入Docker Network命名空间配置

3.15.1　问题

你希望更深入地理解 Docker Network（参见范例 3.14）是如何工作的，尤其是 VXLAN 网络接口都可以在哪些场景下使用。

3.15.2　解决方案

新的 Docker Network 中覆盖网络利用了 VXLAN 隧道和网络命名空间技术。在范例 3.7 中，你已经学习了如何查看和管理网络命名空间。在 Docker Network 中，你也可以进行同样的操作。

本范例涉及的是比较高级的话题，你可以更深入地理解 Docker 如何利用网络命名空间来构建一个 VXLAN 覆盖网络。如果你只是使用默认的单主机 Docker 网络或者多主机 Docker Network，或者其他在范例 3.11 和范例 3.13 中介绍的多主机网络解决方案，那么本范例并不是必须要掌握的。

你可以在 Docker Network 设计文档（https://github.com/docker/libnetwork/blob/master/docs/design.md）中获得更详细的说明。但是为了更好地解释这些概念，没有什么比一个实际的例子效果更好了。

3.15.3　讨论

对于一个在覆盖网络中运行的容器，可以像下面这样查看它的网络命名空间。

```
$ docker inspect -f '{{ .NetworkSettings.SandboxKey}}' 6635a3822135
/var/run/docker/netns/6635a3822135
```

这并不是网络命名空间的默认存储位置，这看起来可能有一些迷惑性。因此，让我们先切换为 root 用户，进入这个容器网络命名空间所在的目录，查看里面的网络接口，如下所示。

```
$ sudo su
root@net-2:/home/vagrant# cd /var/run/docker/
root@net-2:/var/run/docker# ls netns
6635a3822135
8805e22ad6e2
```

要想使用 `ip` 命令来查看这个网络命名空间中的网络设备接口，我们先创建一个指向 /var/run/docker/netns 的文件链接 `netns`，如下所示。

```
root@net-2:/var/run# ln -s /var/run/docker/netns netns
root@net-2:/var/run# ip netns show
6635a3822135
8805e22ad6e2
```

上面的命令会返回两个网络命名空间的 ID，其中一个是正在运行中的该 Docker 主机上的容器，另一个则是这个容器所在的覆盖网络的 ID。

```
root@net-2:/var/run/docker# ip netns exec 6635a3822135 ip addr show eth0
15: eth0: <BROADCAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:b3:91:22:c3 brd ff:ff:ff:ff:ff:ff
    inet 172.21.0.5/16 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::42:b3ff:fe91:22c3/64 scope link
       valid_lft forever preferred_lft forever
```

回到运行中的容器，查看一下它的网络接口设备，你会发现它们的 MAC 地址和 IP 地址是相同的。如果查看一下覆盖网络命名空间中的链路信息，你会看到一个 VXLAN 设备接口以及它所使用的 VLAN ID，如下所示。

```
root@net-2:/var/run/docker# ip netns exec 8805e22ad6e2 ip -d link show
...<snip>...
14: vxlan1: <BROADCAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master br0 state \
UNKNOWN mode DEFAULT group default
    link/ether 7a:af:20:ee:e3:81 brd ff:ff:ff:ff:ff:ff promiscuity 1
    vxlan id 256 srcport 32768 61000 dstport 8472 proxy l2miss l3miss ageing 300
    bridge_slave
16: veth2: <BROADCAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast master br0 state \
UP mode DEFAULT group default qlen 1000
    link/ether 46:b1:e2:5c:48:a8 brd ff:ff:ff:ff:ff:ff promiscuity 1
    veth
    bridge_slave
```

如果抓取一下这些网络接口上的数据包，你将会看到容器之间的通信数据。

第 4 章

开发和配置Docker

4.0 简介

如果你已经阅读了本书前面的全部章节，那么应该已经学习了有关 Docker 用法的所有基本知识。你已经可以安装 Docker 引擎，创建和管理容器，构建和共享镜像，并且已经很好地理解了容器的网络模型，包括跨主机的容器网络。本章将会学习一些更高级的 Docker 主题：首先是面向开发人员的内容，其次是介绍如何对 Docker 进行配置。

在范例 4.1 中，我们会看一下如何对 Docker 引擎进行配置，然后在范例 4.2 中会向你介绍如何从源代码编译 Docker。范例 4.3 将会介绍如何运行所有的测试来验证你的构建，范例 4.4 则会介绍如何使用最新构建出来的 Docker 可执行文件代替官方发布的 Docker 引擎。

开发人员可能还想看一下 `nsenter` 工具，范例 4.5 会对这一工具进行介绍。尽管仅使用 Docker 并不要求你掌握这一工具，但是这个工具可以帮助你更好地理解 Docker 是如何利用 Linux 命名空间来创建容器的。通过范例 4.6 可以更深入地了解用于管理容器的底层系统库。最初被称为 libcontainer 的 `runc` 项目，已经作为参考实现贡献给了开放容器项目（Open Container Initiative），以帮助推动容器运行时和镜像格式的标准化。

为了更深入地对容器进行配置以及控制如何访问容器引擎，范例 4.7 中将会介绍如何远程访问 Docker 守护进程。范例 4.8 中将会介绍 Docker 提供的 API，Docker 客户端通过这个 API 来管理容器。范例 4.9 中将会介绍如何从远程安全地访问 Docker API，讲解如何对基于 TLS 通信方式访问 Docker 引擎进行设置。范例 4.12 是关于 Docker 配置内容的最后一个主题。该范例将会介绍如何更改底层存储驱动程序，这是一个可以支持 Docker 镜像的联合文件系统。

如果你是一个 Docker 用户，那么范例 4.10 和范例 4.11 应该会让你受益匪浅。这两个范例

主要介绍了 docker-py，这是一个用于与 Docker API 进行通信的 Python 模块。在 Docker 中，这并不是你可以使用的唯一客户端，但它是学习 Docker API 的简单的入门途径。

4.1 管理和配置Docker守护进程

4.1.1 问题

你想启动、停止和重新启动 Docker 守护进程。此外，你还想对 Docker 守护进程进行不同的配置，比如 Docker 可执行程序的位置，或者使用一个不同的网桥设备。

4.1.2 解决方案

使用 docker 初始化脚本来管理 Docker 守护进程。在大多数基于 Ubuntu 或 Debian 的系统上，这个脚本位于目录 /etc/init.d/docker 下。像很多其他初始化服务一样，你可以通过 service 命令来管理 Docker 服务。Docker 守护进程以 root 用户身份运行，如下所示。

```
# service docker status
docker start/running, process 2851
# service docker stop
docker stop/waiting
# service docker start
docker start/running, process 3119
```

配置文件位于目录 /etc/default/docker 下。在 Ubuntu 系统上，所有的配置变量都被注释掉了。/etc/default/docker 文件的内容如下所示。

```
# Docker Upstart and SysVinit configuration file

# Customize location of Docker binary (especially for development testing).
#DOCKER="/usr/local/bin/docker"

# Use DOCKER_OPTS to modify the daemon startup options.
#DOCKER_OPTS="--dns 8.8.8.8 --dns 8.8.4.4"

# If you need Docker to use an HTTP proxy, it can also be specified here.
#export http_proxy="http://127.0.0.1:3128/"

# This is also a handy place to tweak where Docker's temporary files go.
#export TMPDIR="/mnt/bigdrive/docker-tmp"
```

例如，如果你想配置守护进程对 TCP 套接字进行监听以启用远程 API 访问，就应该像范例 4.7 中介绍的那样编辑这个配置文件。

4.1.3 讨论

在基于 systemd 的系统（比如 Ubuntu 15.05 或者 CentOS 7）上，你需要修改 Docker 的 systemd 单元文件。这个文件可能位于 /etc/systemd/system/docker.service.d 文件夹中，也可能是 /etc/systemd/system/docker.service 文件。要想了解关于如何在 systemd 下对 Docker 守

护进程进行配置的详细信息，可以参考 Docker 官方文档中的文章（https://docs.docker.com/articles/systemd/）。

最后，尽管可以通过 Linux 守护进程的方式来运行 Docker，但你仍可以通过 `docker -d` 命令以交互式方式启动 Docker 守护进程，或者在 Docker 1.8 之后，你也可以使用 `docker daemon` 命令。这种情况下可以直接在 Docker 命令后面添加命令行参数。查看 Docker 命令的帮助信息，可以确认一下有哪些选项可以使用，如下所示。

```
$ docker daemon --help

Usage:  docker daemon [OPTIONS]

Enable daemon mode

 --api-cors-header=                     Set CORS headers in the remote API
 -b, --bridge=                          Attach containers to a network bridge
 --bip=                                 Specify network bridge IP
 -D, --debug=false                      Enable debug mode
 --default-gateway=                     Container default gateway IPv4 address
 ...
```

4.2　从源代码编译自己的Docker二进制文件

4.2.1　问题

你想开发 Docker 软件并构建自己的 Docker 二进制文件。

4.2.2　解决方案

使用 Git 从 GitHub 克隆 Docker 的代码仓库（https://github.com/docker/docker），然后通过 Makefile 创建自己的二进制文件。

Docker 程序是在一个 Docker 容器中构建的。在一台 Docker 主机中，你可以克隆 Docker 的源代码仓库，然后使用 Makefile 构建一个新的二进制文件。

这个新的 Docker 二进制文件通过运行一个特权模式的 Docker 容器获得。这个 Makefile 文件包含几个构建目标，其中包括一个 `binary` 目标，如下所示。

```
$ cat Makefile
...
default: binary

all: build
  $(DOCKER_RUN_DOCKER) hack/make.sh

binary: build
  $(DOCKER_RUN_DOCKER) hack/make.sh binary
...
```

因此，只需要简单地执行 `sudo make binary` 即可。

Docker 项目根目录底下的 hack 已经被移动到了 project 文件夹中。所以实际上，make.sh 脚本就是 project/make.sh。这个文件会使用位于 project/make/ 文件夹下的脚本来完成各种构建任务。

```
$ sudo make binary
...
docker run --rm -it --privileged \
                    -e BUILDFLAGS -e DOCKER_CLIENTONLY -e DOCKER_EXECDRIVER \
                    -e DOCKER_GRAPHDRIVER -e TESTDIRS -e TESTFLAGS \
                    -e TIMEOUT \
                    -v "/tmp/docker/bundles:/go/src/github.com/docker/docker/\
                                            bundles" \
                    "docker:master" hack/make.sh binary

---> Making bundle: binary (in bundles/1.9.0.-dev/binary)
Created binary: \
/go/src/github.com/docker/docker/bundles/1.9.0-dev/binary/docker-1.9.0-dev
```

可以看到，Makefile 文件中的 `binary` 目标将会从镜像 docker:master 启动一个特权模式的 Docker 容器，并设置一系列的环境变量，从本地挂载一个卷，以及调用 `hack/make.sh binary` 命令。

在目前的 Docker 开发状态下，新构建的二进制文件将会保存到 bundles/1.9.0-dev/binary/ 文件夹下。实际上，根据 Docker 发布的版本不同，你看到的版本号也可能会略有不同。

4.2.3　讨论

为了简化这一过程，你可以克隆本书附带的代码仓库。本范例提供了一个 Vagrantfile，它会启动一个 Ubuntu 14.04 虚拟机，并在这个虚拟机中安装最新版的 Docker，然后克隆 Docker 源代码仓库，如下所示。

```
$ git clone https://github.com/how2dock/docbook
$ cd docbook/ch04/compile/
$ vagrant up
```

当这个虚拟机启动之后，通过 `ssh` 连接到虚拟机，然后进入到 /tmp/docker 文件夹，这个文件夹是 Vagrant 初始化虚拟机时创建的。在这个文件夹下执行 `make` 命令。第一次执行这个 Makefile 文件的时候，安装在宿主机上的稳定版的 Docker 会拉取构建 Docker 过程所需要的基础镜像 ubuntu:14.04，然后通过 /tmp/docker/Dockerfile 文件构建镜像 docker:master。第一次执行该操作时，可能会花费较长的时间，如下所示。

```
$ vagrant ssh
$ cd /tmp/docker
$ sudo make binary
docker build -t "docker:master" .
Sending build context to Docker daemon 55.95 MB
Sending build context to Docker daemon
Step 0 : FROM ubuntu:14.04
...
```

当上述操作完成之后，你将会得到一个新的 Docker 二进制文件，如下所示。

```
$ cd bundles/1.9.0-dev/binary/docker
$ ls
docker  docker-1.9.0-dev  docker-1.9.0-dev.md5  docker-1.9.0-dev.sha256
```

4.2.4 参考

- 如何在 GitHub 上向 Docker 贡献代码（https://github.com/docker/docker/blob/master/CONTRIBUTING.md）

4.3 为开发Docker运行Docker测试集

4.3.1 问题

你对 Docker 源代码进行了一些修改并且构建了一个新的二进制文件。你还需要确保你能够通过所有测试。

4.3.2 解决方案

使用 Makefile 中的 `test` 目标来运行 Docker 源代码提供的四组测试。或者，只选择那些你关心的测试，如下所示。

```
$ cat Makefile
...
test: build
  $(DOCKER_RUN_DOCKER) hack/make.sh binary cross \
                                    test-unit test-integration \
                                    test-integration-cli test-docker-py
test-unit: build
  $(DOCKER_RUN_DOCKER) hack/make.sh test-unit

test-integration: build
  $(DOCKER_RUN_DOCKER) hack/make.sh test-integration

test-integration-cli: build
  $(DOCKER_RUN_DOCKER) hack/make.sh binary test-integration-cli

test-docker-py: build
  $(DOCKER_RUN_DOCKER) hack/make.sh binary test-docker-py
...
```

可以通过查看 Makefile 文件来选择你需要运行的测试集。如果通过 `make test` 命令运行所有的测试集，该命令也会同时构建 Docker 二进制文件，如下所示。

```
$ sudo make test
....
---> Making bundle: test-docker-py (in bundles/1.9.0-dev/test-docker-py)
+++ exec docker --daemon --debug --storage-driver vfs \
                -exec-driver native \
```

```
                --pidfile \
                /go/src/github.com/docker/docker/bundles/1.9.0-dev/ \
                test-docker-py/docker.pid
........................................................
----------------------------------------------------------------------
Ran 56 tests in 75.366s

OK
```

根据测试覆盖率，如果所有的测试都通过，那么你应该相信新的二进制能正常工作。

4.3.3　参考

- Docker 官方开发环境文档（https://docs.docker.com/project/software-required/）

4.4　使用新的Docker二进制文件替换当前的文件

4.4.1　问题

按照范例 4.2 和范例 4.3 中的介绍，你构建了一个新的 Docker 二进制文件，并通过了单元测试和集成测试。现在你想在你的宿主机上使用这个新的二进制文件。

4.4.2　解决方案

让我们从在范例 4.2 中创建的虚拟机开始。

首先需要停止运行中的 Docker 守护进程。在 Ubuntu 14.04 上，编辑 /etc/default/docker 文件，去掉 `DOCKER` 变量前面的注释，这个变量定义了 Docker 二进制文件的位置。将这个变量设置为 `DOCKER="/usr/local/bin/docker"`。将新的 Docker 二进制文件复制到 /usr/local/bin/docker，然后重新启动 Docker 守护进程，如下所示。

```
$ pwd
/tmp/docker
$ sudo service docker stop
docker stop/waiting
$ sudo vi /etc/default/docker
$ sudo cp bundles/1.8.0-dev/binary/docker-8.0-dev /usr/local/bin/docker
$ sudo cp bundles/1.8.0-dev/binary/docker-1.8.0-dev /usr/bin/docker
$ sudo service docker restart
stop: Unknown instance:
$ docker version
Client:
 Version:      1.8.0-dev
 API version:  1.21
 Go version:   go1.4.2
 Git commit:   3e596da
 Built:        Tue Aug 11 16:51:56 UTC 2015
 OS/Arch:      linux/amd64

Server:
```

```
Version:      1.8.0-dev
API version:  1.21
Go version:   go1.4.2
Git commit:   3e596da
Built:        Tue Aug 11 16:51:56 UTC 2015
OS/Arch:      linux/amd64
```

现在你使用的就是主开发分支上最新版的 Docker（即本书编写时主分支的 Git 提交号 3e596da）。

4.4.3　讨论

在 Vagrant 虚拟机中的 Docker 初始化脚本通过下面的方式安装了最新的稳定版 Docker，如下所示。

```
sudo curl -sSL https://get.docker.com/ubuntu/ | sudo sh
```

这种安装方式会将 Docker 二进制文件保存到 /usr/bin/docker。这可能会与你新构建的 Docker 二进制文件冲突。当运行 `docker version` 命令时，如果你发现两个二进制文件有冲突，你可以删除这个文件或者使用新的二进制文件替换该文件。

4.5　使用nsenter

4.5.1　问题

你希望能进入到容器进行调试，你运行的 Docker 版本低于 1.3.1，或者你不想使用 `docker exec` 命令。

4.5.2　解决方案

使用 `nsenter`（https://github.com/jpetazzo/nsenter）。从 Docker 1.3 开始，你可以使用 `docker exec` 轻松进入一个运行中的容器，所以已经没必要在容器中运行 SSH 服务并暴露 22 端口，或者使用已经不再推荐使用的 `attach` 命令。

`nsenter` 诞生于 `docker exec` 之前，用于解决如何进入容器命名空间（因此得名 nsenter）的问题。尽管如此，`nsenter` 也是一个很实用的工具，本书中我们也为其准备了一个简短的范例。

我们将会启动一个容器，这个容器会睡眠一段时间。考虑到完整性，让我们使用 `docker exec` 命令进入这个运行中的容器，如下所示。

```
$ docker pull ubuntu:14.04
$ docker run -d --name sleep ubuntu:14.04 sleep 300
$ docker exec -ti sleep bash
root@db9675525fab:/#
```

`nsenter` 也可以完成同样的工作。而且很方便的是，你可以在 Docker Hub 上找到 `nsenter` 的镜像。下载这个镜像，启动一个容器，然后运行 `nsenter`。

```
$ docker pull jpetazzo/nsenter
$ sudo docker run docker run --rm -v /usr/local/bin:/target jpetazzo/nsenter
```

现在，了解一下 `nsenter` 镜像的 Dockerfile（https://github.com/jpetazzo/nsenter/blob/master/Dockerfile）及其 `CMD` 参数将会十分有用。你会看到它运行了一个 installer 脚本。这个简单的脚本除了可以检测是否存在 /target 挂载点之外，没有其他任何作用。如果这个挂载点存在，该脚本就会将脚本文件 docker-enter 和二进制文件 nsenter 复制到这个挂载点。在 `docker run` 命令中，由于你指定了一个卷（即 `-v /usr/local/bin:/target`），因此运行这个容器会将 nsenter 文件复制到你的本地主机。这是一个很巧妙的技巧，并且取得了很好的效果，如下所示。

```
$ which docker-enter nsenter
/usr/local/bin/docker-enter
/usr/local/bin/nsenter
```

为了将文件复制到 /usr/local/bin 下面，在运行容器时我使用了 `sudo`。如果不想使用默认的挂载点，你可以使用类似下面的命令将这两个文件复制到本地主机上，如下所示。

```
$ docker run --rm jpetazzo/nsenter cat /nsenter \
                  > /tmp/nsenter && chmod +x /tmp/nsenter
```

现在你就可以进入容器内部了。如果不想使用容器内的交互式 shell，你也可以指定想要在容器中运行的命令，如下所示。

```
$ docker-enter sleep
root@db9675525fab:/#
$ docker-enter sleep hostname
db9675525fab
```

`docker-enter` 是对 `nsenter` 的一个包装。你可以在通过 `docker inspect` 命令找到容器的进程 ID 之后，直接使用 `nsenter` 命令，如下所示。

```
$ docker inspect --format {{.State.Pid}} sleep
9302
$ sudo nsenter --target 9302 --mount --uts --ipc --net --pid
root@db9675525fab:/#
```

4.5.3　讨论

从 Docker 1.3 开始，我们可以使用 `docker exec` 来代替 `nsenter`，如下所示。

```
$ docker exec -h

Usage: docker exec [OPTIONS] CONTAINER COMMAND [ARG...]

Run a command in a running container

  -d, --detach=false         Detached mode: run command in the background
  --help=false               Print usage
  -i, --interactive=false    Keep STDIN open even if not attached
  -t, --tty=false            Allocate a pseudo-TTY
```

4.5.4 参考

- GitHub 上 Jerome Petazzoni 的 `nsenter` 代码仓库（https://github.com/jpetazzo/nsenter）

4.6 `runc`简介

4.6.1 问题

你希望熟悉即将发布的关于容器格式的标准，以及容器运行时 `runc`。

4.6.2 解决方案

开放容器项目（Open Container Project，OCP）设立于 2015 年 6 月，该项目的规范正在制定中，目前还没有完成。

但是，Docker 公司捐赠了它们的 libcontainer（https://github.com/docker/libcontainer）代码，这是关于容器运行时标准的早期实现。这个运行时被称为 `runc`。

> OCP 刚成立不久，该规范还没有完成。在其参考实现被认为稳定并且成熟之前，该规范可能还会有很多修改。

本范例将会为你带来关于 `runc` 的直观感受，包括编译 Go 代码库的过程。像其他章节一样，这里我也准备了一个可以作为 Docker 主机的 Vagrant 虚拟机，里面还有 Go 1.4.2，以及 `runc` 代码的克隆。你可以通过下面的命令启动这个虚拟机。

```
$ git clone https://github.com/how2dock/docbook.git
$ cd docbook/ch04/runc
$ vagrant up
$ vagrant ssh
```

在进入到这个虚拟机的终端之后，你需要先通过 `go get` 来下载 `runc` 的所有依赖。当依赖下载完毕之后，就可以开始构建和安装 `runc` 了。请确保在你的系统路径中能找到 `runc`。

> 估计不久之后该构建过程就会发生改变。可能会使用 Docker 本身进行构建。

```
$ cd go/src
$ go get github.com/opencontainers/runc
$ cd github.com/opencontainers/runc/
$ make
```

```
$ sudo make install
$ runc -v
runc version 0.2
```

要想通过 runc 来运行一个容器，你需要一个根文件系统来描述你的容器镜像。获得容器镜像的最简单的方式是，使用 Docker 本身并使用 docker export 命令。所以我们来拉取一个镜像，启动一个容器后将它导出为一个压缩包，如下所示。

```
$ cd ~
$ mkdir foobar
$ cd foobar
$ docker run --name foobar -d ubuntu:14.04 sleep 30
$ docker export -o foobar.tar foobar
$ sudo -xf foobar.tar
$ rm foobar.tar
```

为了运行这个容器，你需要生成一个配置文件。这可以非常方便地通过 runc spec 命令来完成。为了快速启动一个容器，你只需要修改一个地方，那就是根文件系统的位置。在这个 JSON 文件中，编辑 path 属性。下面是这个配置文件的部分摘要。

```
$ runc spec > config.json
$ vi config.json
...
  "root": {
    "path": "./",
    "readonly": true
...
```

现在你就可以以 root 用户身份执行 runc 命令来启动你的容器了，之后将会进入容器中的 shell，如下所示。

```
$ sudo runc
#
```

这是对 Docker 和开放容器标准将会如何发展的一个更深层次的探索。你现在可以浏览一下配置文件内容，看看如何为你的容器定义启动命令、网络命名空间和各种卷挂载。

4.6.3 讨论

开放容器项目是一个好消息。2014 年末，CoreOS 已经开始开发容器镜像的开放标准 appc（https://github.com/appc/spec），包括一个新的信任机制。CoreOS 也开发了一个可以运行基于 appc 的容器运行时实现。作为 OCP 的一部分，appc 开发人员将会帮助制定新的 runc 规范，从而避免容器镜像格式与运行时实现的分裂。

如果看一下应用程序容器镜像（application container image，ACI）（https://github.com/coreos/rkt/blob/master/Documentation/app-container.md#ACI）的清单，你会发现，这与前面解决方案部分中 runc spec 创建的配置文件非常相似。你会看到一些 rkt 实现的功能也会被移植到 runc 中。

如果你关心容器标准，可以关注一下开放容器项目，等待容器标准的发布。

4.6.4　参考

- cloudgear.net 上的一篇博客文章（https://www.cloudgear.net/blog/2015/getting-started-with-runc/）启发了本范例
- 开放容器项目（https://www.opencontainers.org）
- 应用容器规范（https://github.com/appc/spec）
- `rkt` 运行时（https://github.com/coreos/rkt）

4.7　远程访问Docker守护进程

4.7.1　问题

默认的 Docker 守护进程监听 Unix 套接字（`/var/run/docker.sock`），因此你只能在本地主机上访问到 Docker 守护进程。但是，你想远程访问 Docker 主机，在不同的主机上调用 Docker 的 API。

4.7.2　解决方案

修改配置文件 /etc/default/docker 来切换 Docker 守护进程的监听协议，启用远程 API 调用。

在 /etc/default/docker 文件中，添加一行设置 `DOCKER_HOST` 使用 `tcp`，端口为 2375。然后通过 `sudo service docker restart` 命令重新启动 Docker 守护进程，如下所示。

```
$ cat /etc/default/docker
...
# Use DOCKER_OPTS to modify the daemon startup options.
#DOCKER_OPTS="--dns 8.8.8.8 --dns 8.8.4.4"
DOCKER_OPTS="-H tcp://127.0.0.1:2375"
...
```

之后你就可以使用 Docker 客户端通过 TCP 来访问指定远程主机，如下所示。

```
$ docker -H tcp://127.0.0.1:2375 images
REPOSITORY      TAG           IMAGE ID        CREATED         VIRTUAL SIZE
ubuntu          14.04         04c5d3b7b065    6 days ago      192.7 MB
```

这种方法没有使用加密和身份验证机制。你不应该在一台通过公网可以访问的主机上使用这样的配置，否则你的 Docker 守护进程将会暴露给所有人。如果想在生产环境中从远程访问 Docker 守护进程，你需要确保 Docker 守护进程的安全性（参见范例 4.9）。

4.7.3 讨论

Docker 守护进程监听在 TCP 协议之后，你现在可以使用 `curl` 发起 API 调用并查看返回结果。这是一种非常高效的学习 Docker 远程 API 的途径，如下所示。

```
$ curl -s http://127.0.0.1:2375/images/json | python -m json.tool
[
    {
        "Created": 1418673175,
        "Id": "04c5d3b7b0656168630d3ba35d8889bdaafcaeb32bfbc47e7c5d35d2",
        "ParentId": "d735006ad9c1b1563e021d7a4fecfd384e2a1c42e78d8261b83d6271",
        "RepoTags": [
            "ubuntu:14.04"
        ],
        "Size": 0,
        "VirtualSize": 192676726
    }
]
```

我们将 `curl` 命令的输出通过管道的方式传递给了 `python -m json.tool`，让结果的 JSON 对象更易读。`-s` 参数用于隐藏数据传输过程中的调试信息。

4.8 通过Docker远程API完成自动化任务

4.8.1 问题

在可以远程访问 Docker 守护进程之后（参见范例 4.7），你希望浏览一下 Docker 远程 API 以便编写程序。你可以使用这些 API 进行 Docker 任务的自动化。

4.8.2 解决方案

Docker 远程 API 提供了详细的文档（https://docs.docker.com/reference/api/docker_remote_api_v1.20/）。现在的远程 API 版本为 1.20。Docker 远程 API 是一个 REST API，通过使用不同的 HTTP 方法（比如 `GET`、`POST` 和 `DELETE`）进行 HTTP 调用，对各种资源（比如镜像和容器）进行管理。attach 和 pull 这两个 API 不是纯 REST 的，这在 API 文档中也有说明。

你已经学习了如何让 Docker 守护进程监听 TCP 套接字（参见范例 4.7），并使用 `curl` 进行 API 调用。表 4-1 和表 4-2 总结了现在能使用的 Docker 远程 API。

表4-1：对容器进行操作的API

容器操作	HTTP方法	URI	容器操作	HTTP方法	URI
获取容器列表	`GET`	/containers/json	重启容器	`POST`	/containers/(id)/restart
创建容器	`POST`	/containers/create	终止容器	`POST`	/containers/(id)/kill
查看容器详情	`GET`	/containers/(id)/json	暂停容器	`POST`	/containers/(id)/pause
启动容器	`POST`	/containers/(id)/start	删除容器	`DELETE`	/containers/(id)
停止容器	`POST`	/containers/(id)/stop			

表4-2：对镜像进行操作的API

镜像操作	HTTP方法	URI
获取镜像列表	GET	/images/json
创建镜像	POST	/images/create
在镜像仓库中为镜像打标签	POST	/images/(name)/tag
删除镜像	DELETE	/images/(name)
查找镜像	GET	/images/search

比如，让我们从公有 registry（即 Docker Hub）下载 Ubuntu 14.04 的镜像，从这个镜像创建一个容器并启动这个容器，然后删除这个容器和镜像。注意，这只是一个示例而已，这个容器开始运行之后会立刻退出，因为你没有指定任何要在容器中运行的命令，如下所示。

```
$ curl -X POST -d "fromImage=ubuntu" -d "tag=14.04"
                                        http://127.0.0.1:2375/images/create
$ curl -X POST -H 'Content-Type: application/json'
               -d '{"Image":"ubuntu:14.04"}'
         http://127.0.0.1:2375/containers/create
{"Id":"6b6bd46f483a5704d4bced62ff58a0ac5758fb0875ec881fa68f0e...",\
"Warnings":null}
$ docker ps
CONTAINER ID    IMAGE           COMMAND         CREATED         STATUS   ...
$ docker ps -a
CONTAINER ID    IMAGE           COMMAND         CREATED         STATUS   ...
6b6bd46f483a    ubuntu:14.04    "/bin/bash"     16 seconds ago           ...
$ curl -X POST http://127.0.0.1:2375/containers/6b6bd46f483a/start
$ docker ps -a
CONTAINER ID    IMAGE           COMMAND         CREATED              ...
6b6bd46f483a    ubuntu:14.04    "/bin/bash"     About a minute ago   ...
```

现在让我们来做清理工作，如下所示。

```
$ curl -X DELETE http://127.0.0.1:2375/containers/6b6bd46f483a
$ curl -X DELETE http://127.0.0.1:2375/images/04c5d3b7b065
[{"Untagged":"ubuntu:14.04"}
,{"Deleted":"04c5d3b7b0656168630d3ba35d8889bd0e9caafcaeb3004d2bfbc47e7c5d35d2"}
,{"Deleted":"d735006ad9c1b1563e021d7a4fecfd75ed36d4384e2a1c42e78d8261b83d6271"}
,{"Deleted":"70c8faa62a44b9f6a70ec3a018ec14ec95717ebed2016430e57fec1abc90a879"}
,{"Deleted":"c7b7c64195686444123ef370322b5270b098c77dc2d62208e8a9ce28a11a63f9"}
,{"Deleted":"511136ea3c5a64f264b78b5433614aec563103b4d4702f3ba7d4d2698e22c158"}
$ docker ps -a
CONTAINER ID    IMAGE           COMMAND         CREATED         STATUS   ...
$ docker images
REPOSITORY      TAG             IMAGE ID        CREATED         VIRTUAL SIZE
```

在启用远程 API 访问之后，你可以设置环境变量 DOCKER_HOST 指向远程 Docker 守护进程的 HTTP 端点。这样在执行 docker 命令的时候就不需要再使用 -H 选项了。比如，对命令 docker -H http://127.0.0.1:2375 ps 来说，你可以运行 export DOCKER_HOST=tcp://127.0.0.1:2375，这样就可以直接执行 docker ps 了。

4.8.3 讨论

尽管可以使用 `curl` 命令或者编写自己的客户端，但是使用已有的 Docker 客户端软件可以方便你进行 API 调用，比如 docker-py（参见范例 4.10）。

表 4-1 和表 4-2 并没有列出完整的 API，你应该到官方文档（https://docs.docker.com/reference/api/docker_remote_api_v1.20/）查看所有 API 以及它们的参数和返回结果示例。

4.9 从远程安全访问Docker守护进程

4.9.1 问题

你需要安全地远程访问 Docker 守护进程。

4.9.2 解决方案

为 Docker 守护进程配置基于 TLS（http://tools.ietf.org/html/rfc5246）的访问。这将使用公共密钥加密来对 Docker 客户端和启用了 TLS 的 Docker 守护进程之间的通信进行加密和身份验证。

Docker 官方文档（https://docs.docker.com/articles/https/#daemon-modes）记载了如何测试这一安全特性的基本步骤。但是，这个例子介绍了如何创建自己的证书颁发机构（certificate authority，CA），并用这个 CA 来对服务器端和客户端的证书进行签名。在正规的基础设施中，你应该联系你熟悉的 CA，并从这个 CA 获得一个服务器端证书。

为了方便测试 TLS 配置，我创建了一个 Docker 镜像（https://registry.hub.docker.com/u/runseb/dockertls/），这个镜像提供了一个脚本，用于创建 CA 以及服务器端和客户端的证书和密钥。你可以从这个镜像启动一个容器，创建测试所需要的各种文件。

我们将使用 Ubuntu 14.04 主机，并运行最新版的 Docker（参见范例 1.1）。下载上面提到的镜像之后，启动一个容器。你需要将本地卷挂载到容器内的 /tmp/ca 上。你还需要在启动容器的时候指定主机名（下面例子中的）。当这个容器运行结束之后，所有的 CA、服务器端和客户端的密钥以及证书都会保存到当前工作目录下，如下所示。

```
$ docker pull runseb/dockertls
$ docker run -ti -v $(pwd):/tmp/ca runseb/dockertls <hostname>
$ ls
cakey.pem  ca.pem  ca.srl  clientcert.pem  client.csr  clientkey.pem
extfile.cnf  makeca.sh  servercert.pem  server.csr  serverkey.pem
```

停止正在运行中的 Docker 守护进程。创建一个 /etc/docker 的文件夹以及一个 ~/.docker 文件夹。将 CA、服务器端密钥和证书复制到 /etc/docker 文件夹下。将 CA、客户端密钥和证书复制到 ~/.docker 文件夹下，如下所示。

```
$ sudo service docker stop
$ sudo mkdir /etc/docker
$ mkdir ~/.docker
$ sudo cp {ca,servercert,serverkey}.pem /etc/docker
```

```
$ cp ca.pem ~/.docker/
$ cp clientkey.pem ~/.docker/key.pem
$ cp clientcert.pem ~/.docker/cert.pem
```

编辑 /etc/default/docker 配置文件（这需要 root 身份），修改 `DOCKER_OPTS`（将 `test` 替换为你自己的主机名），其内容如下所示。

```
DOCKER_OPTS="-H tcp://<test>:2376 --tlsverify \
                                  --tlscacert=/etc/docker/ca.pem \
                                  --tlscert=/etc/docker/servercert.pem \
                                  --tlskey=/etc/docker/serverkey.pem"
```

然后通过 `sudo service docker restart` 重新启动 Docker 服务，尝试连接 Docker 守护进程，如下所示。

```
$ docker -H tcp://test:2376 --tlsverify images
REPOSITORY          TAG             IMAGE ID        CREATED         VIRTUAL SIZE
runseb/dockertls    latest          5ed60e0f6a7c    17 minutes ago  214.7 MB
```

4.9.3 讨论

runseb/dockertls 是一个自动构建的镜像，其 Dockerfile 可以在 https://github.com/how2dock/docbook/ch04/tls 上查看。

通过设置一些环境变量（`DOCKER_HOST` 和 `DOCKER_TLS_VERIFY`），你可以在 CLI 中轻松配置 TLS 连接，如下所示。

```
$ export DOCKER_HOST=tcp://test:2376
$ export DOCKER_TLS_VERIFY=1
$ docker images
REPOSITORY          TAG             IMAGE ID        CREATED         VIRTUAL SIZE
runseb/dockertls    latest          5ed60e0f6a7c    19 minutes ago  214.7 MB
```

范例 4.7 中介绍的 `curl` 命令仍然可以使用，但是你需要指定客户端的密钥和证书，如下所示。

```
$ curl --insecure --cert ~/.docker/cert.pem --key ~/.docker/key.pem \
       -s https://test:2376/images/json | python -m json.tool
[
    {
        "Created": 1419280147,
        "Id": "5ed60e0f6a7ce3df3614d20dcadf2e4d43f4054da64d52709c1559ac",
        "ParentId": "138f848eb669500df577ca5b7354cef5e65b3c728b0c241221c611b1",
        "RepoTags": [
            "runseb/dockertls:latest"
        ],
        "Size": 0,
        "VirtualSize": 214723529
    }
]
```

需要注意的是，由于你使用的是自己创建的证书颁发机构，所以这里使用了 `curl` 命令的 `--insecure` 选项。默认情况下，`curl` 命令将会向宿主机上安装的 CA 列表中的 CA 进行证书的验证。如果服务器端以及客户端的密钥和证书来自宿主机中已经安装的 CA 列表中的某一 CA，则可以不用再建立 `--insecure` 的连接。不过，使用这个选项并不代表该 TLS 不能正常工作。

4.10 使用docker-py访问远程Docker守护进程

4.10.1 问题

尽管 Docker 客户端程序很强大，但是你希望利用 Python 客户端软件来访问 Docker 守护进程。具体来说，你想编写一个 Python 程序来调用 Docker 远程 API。

4.10.2 解决方案

通过 Pip 安装 docker-py 模块。在一个 Python 脚本或者交互式 shell 中，创建一个远程 Docker 守护进程连接，开始 API 调用。

> 尽管本范例主要讲 docker-py，但是这个例子也说明了你可以使用自己的客户端软件与 Docker 守护进程交互，不必局限于默认的 Docker 客户端。有很多使用各种编程语言（https://docs.docker.com/reference/api/remote_api_client_libraries/）编写的客户端程序，比如 Java、Groovy、Perl、PHP、Scala 和 Erlang 等，你也可以通过学习 Docker API 参考文档（https://docs.docker.com/reference/api/docker_remote_api_v1.16/）来编写自己的 Docker 客户端。

docker-py 是一个由 Python 编写的 Docker 客户端。你可以从源代码（https://github.com/docker/docker-py）进行安装，或者使用更方便的方法——通过 `pip` 命令从 Python Package Index（https://pypi.python.org/pypi）下载该软件。不过，首先你需要安装 `python-pip`，然后再安装 docker-py 包。在 Ubuntu 14.04 上可以进行如下操作。

```
$ sudo apt-get install python-pip
$ sudo pip install docker-py
```

docker-py 的文档（http://docker-py.readthedocs.org/en/latest/）讲述了如何创建一个远程 Docker 守护进程的连接。创建一个 `Client` 类的实例，通过 `base_url` 参数来指定 Docker 守护进程的监听信息。假设 Docker 守护进程监听本地的 Unix 套接字，则创建连接的过程如下所示。

```
$ python
Python 2.7.6 (default, Mar 22 2014, 22:59:56)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> from docker import Client
>>> c=Client(base_url="unix://var/run/docker.sock")
>>> c.containers()
[]
```

假如你的 Docker 主机按照范例 4.7 的说明让 Docker 守护进程监听 TCP，则创建远程连接的方法如下所示。

```
$ python
Python 2.7.6 (default, Mar 22 2014, 22:59:56)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> from docker import Client
>>> c=Client(base_url="tcp://127.0.0.1:2375")
>>> c.containers()
[]
```

你可以在 Python 交互式会话的提示符下输入 `help(c)`，来获得 docker-py 提供的方法列表。

4.10.3　讨论

docker-py 模块有一些基本文档（http://docker-py.readthedocs.org/en/latest/）。其中值得一提的是，该模块通过一个用于创建 Boot2Docker 连接（http://docker-py.readthedocs.org/en/latest/boot2docker/）的帮助方法，提供了与 Boot2Docker（范例 1.7）的集成。由于最新的 Boot2Docker 使用了 TLS 来提供对 Docker 守护进程的安全访问，实际的操作步骤与我们的演示内容可能会有一些不同。不过，对有兴趣试用 docker-py 的用户来说，这个模块还有一个 bug 需要额外提醒一下。

启动 Boot2Docker。

```
$ boot2docker start
Waiting for VM and Docker daemon to start...
................oooo
Started.
Writing /Users/sebgoa/.boot2docker/certs/boot2docker-vm/ca.pem
Writing /Users/sebgoa/.boot2docker/certs/boot2docker-vm/cert.pem
Writing /Users/sebgoa/.boot2docker/certs/boot2docker-vm/key.pem

To connect the Docker client to the Docker daemon, please set:
    export DOCKER_HOST=tcp://192.168.59.103:2376
    export DOCKER_CERT_PATH=/Users/sebgoa/.boot2docker/certs/boot2docker-vm
    export DOCKER_TLS_VERIFY=1
```

上面的命令输出了一组需要进行设置的环境变量。Boot2Docker 提供了一个非常方便的工具 `$(boot2docker shellinit)` 来帮助你进行各种设置。但是，为了让 docker-py 正常工作，你需要编辑 /etc/hosts 文件并设置不同的 `DOCKER_HOST`。在 /etc/hosts 文件中，添加一条记录了 `boot2docker` IP 地址及其本地 DNS 名称（即 `boot2docker`）的记录，然后执行 `export DOCKER_HOST=tcp://boot2docker:2376`。之后在 Python 交互式 shell 中进行如下操作。

```
>>> from docker.client import Client
>>> from docker.utils import kwargs_from_env
>>> client = Client(**kwargs_from_env())
>>> client.containers()
[]
```

4.11　安全使用docker-py

4.11.1　问题

你希望通过 docker-py Python 客户端来访问远程的启用了 TLS 安全访问的 Docker 守护进程。

4.11.2　解决方案

按照范例 4.9 中的步骤设置好 Docker 主机之后，检查一下是否能通过 TLS 连接到 Docker 守护进程。

举例来说，假设你的主机名为 `dockerpytls`，客户端的证书、密钥和 CA 都保存在默认的位置 ~/.docker/，可以尝试如下操作。

```
$ docker -H tcp://dockerpytls:2376 --tlsverify ps
CONTAINER ID    IMAGE       COMMAND     CREATED     STATUS      PORTS       NAMES
```

请确认你已经安装了 docker_py 包，代码如下所示。

```
sudo apt-get -y install python-pip
sudo pip install docker-py
```

当确认 TLS 可以正常工作之后，打开一个 Python 交互式 shell，使用如下的配置创建一个 docker-py 客户端实例。

```
tls_config = docker.tls.TLSConfig(
  client_cert=('/home/vagrant/.docker./cert.pem', \
               '/home/vagrant/.docker/key.pem'), \
  ca_cert='/home/vagrant/.docker/ca.pem')
client = docker.Client(base_url='https://host:2376', tls=tls_config)
```

这段代码等同于在命令行上使用下面的命令对 Docker 守护进程发起调用。

```
$ docker -H tcp://host:2376 --tlsverify --tlscert /path/to/client-cert.pem \
                                       --tlskey /path/to/client-key.pem \
                                       --tlscacert /path/to/ca.pem ...
```

4.11.3　参考

- docker-py 文档（http://docker-py.readthedocs.org/en/latest/tls/）
- Docker 对 HTTPS 的支持（https://docs.docker.com/articles/https/）

4.12　更改存储驱动程序

4.12.1　问题

安装 Docker 时，不想使用系统默认的存储驱动程序，而是使用其他不同的存储驱动程序。

4.12.2 解决方案

本范例将会讲述如何修改 Docker 使用的后端存储。你将会从一台 Ubuntu 14.04 系统开始，该系统的内核版本为 3.13，使用了 AUFS（Another Union File System），并安装了 Docker 1.7，然后你想切换到 overlay 文件系统。像前面的章节一样，你可以使用本书附带的代码仓库中提供的 Vagrantfile。让我们进行如下操作。

```
$ git clone https://github.com/how2dock/docbook.git
$ cd docbook/ch04/overlay
$ vagrant up
$ vagrant ssh
$ uname -r
3.13.0-39-generic
$ docker info | grep Storage
Storage Driver: aufs
$ docker version | grep Server
Server version: 1.7.0
```

Linux 内核自 3.18 之后开始支持 overlay 文件系统。因此 , 为了切换存储驱动程序，你首先需要将你计算机上的内核升级（http://ubuntuhandbook.org/index.php/2014/12/install-linux-kernel-3-18-ubuntu/）到 3.18 版本 , 并重启系统。

```
$ cd /tmp
$ wget http://kernel.ubuntu.com/~kernel-ppa/mainline/v3.18-vivid/\
  linux-headers-3.18.0-031800-generic_3.18.0-031800.201412071935_amd64.deb
$ wget http://kernel.ubuntu.com/~kernel-ppa/mainline/v3.18-vivid/\
  linux-headers-3.18.0-031800_3.18.0-031800.201412071935_all.deb
$ wget http://kernel.ubuntu.com/~kernel-ppa/mainline/v3.18-vivid/\
  linux-image-3.18.0-031800-generic_3.18.0-031800.201412071935_amd64.deb
$ sudo dpkg -i linux-headers-3.18.0-*.deb linux-image-3.18.0-*.deb
$ sudo update-grub
$ sudo shutdown -r now
```

当主机重启完成之后，重新连接到该主机。现在你就可以修改 Docker 的配置文件，启动 Docker 守护进程时通过使用 -s 选项，将存储驱动程序设置为 overlay 文件系统。

```
$ uname -r
3.18.0-031800-generic
$ sudo su
# service docker stop
# echo DOCKER_OPTS=\"-s overlay\" >> /etc/default/docker
# service docker start
```

现在这台主机上 Docker 的存储驱动程序已经切换为 overlay 了。

```
$ docker info | grep Storage
Storage Driver: overlay
```

AUFS 已经是 3.13~3.16 版本内核的默认存储驱动程序了，特别是在 Ubuntu 系统上。从版本 3.18 开始，Linux 上游（主线）内核已经开始支持 overlay 文件系统，而 AUFS 则还没有进入到上游内核。你可以考虑使用 overlay 文件系统。

4.12.3 讨论

Docker 可以使用多种存储后端来存储镜像和容器的文件系统。Docker 存储抽象通过将镜像和容器文件系统保存在层中，并只对层与层之间的变动进行跟踪，以减小存储镜像和容器文件系统所需要的空间。Docker 存储依赖联合文件系统来实现上述功能。

你可以选择下面这些文件系统作为 Docker 的存储后端。

- vfs
- devicemapper
- btrfs
- aufs
- overlay

从 Docker 存储驱动程序的角度来看，分析这些解决方案的稳定性和性能差异，并不在本范例的讨论范围之内。

4.12.4 参考

- 在 Ubuntu 14.04 上将内核升级到 3.18（http://ubuntuhandbook.org/index.php/2014/12/install-linux-kernel-3-18-ubuntu/）
- 深入理解 Docker 存储驱动程序（http://jpetazzo.github.io/assets/2015-03-03-not-so-deep-dive-into-docker-storage-drivers.html#1）
- Docker 支持的文件系统列表（http://www.projectatomic.io/docs/filesystems/）

第 5 章

Kubernetes

5.0 简介

——本节内容由 Joe Beda 提供

随着应用的成长超出单台主机所能承受的负载，对编排系统（orchestration system）的需求也就出现了。编排系统帮助用户将一组主机（也称为节点）视为一个统一、可编程、可靠的集群。这个集群可以当作一台大型计算机来使用。

Kubernetes（有时简称为 k8s，http://kubernetes.io）是 Google 为满足这个需求而开发的开源项目。Kubernetes 的想法来源于 Borg（http://research.google.com/pubs/pub43438.html）和 Omega（http://research.google.com/pubs/pub41684.html），这两个项目在 Google 内部系统中都已经经过了十几年的验证。Google 所有的服务都由这两个系统来运行和管理，包括 Google 搜索、Google 邮件等。构建并运营大规模 Borg 集群的许多工程师也参与到了 Kubernetes 的设计和开发之中。

很长时间以来，Borg 一直都是前 Google 工程师最怀念的东西。但是现在，Kubernetes 为没有在 Google 工作过的工程师弥补了这一缺憾。Kubernetes 也提供了一些增强功能和全新的概念。

5.0.1 增强功能

Kubernetes 集群在多个 Docker 节点之间进行协调，提供一个统一的可编程模型，它具有下面这些增强功能。

- 可靠的容器重启

  Kubernetes 可以监视容器的运行状况，并在出现故障时重新启动容器。

- 自愈

  如果一个节点失效了，Kubernetes 管理系统会自动将失效节点上的任务重新调度到健康的节点上。动态服务归属机制可以确保这些新启动的容器能被发现并使用。

- 高集群利用率

  通过在一组通用的计算机上调度一组不同类型的工作负载，与静态的手动配置方式相比，用户可以大幅提高计算机的利用率。集群越大，工作负荷种类越多，计算机的利用率就越高。

- 组织和分组

  在大型集群中，追踪所有正在运行的容器可能非常困难。Kubernetes 提供了一个灵活的标签（label）系统，让用户和其他系统可以以一组容器为单位来进行处理。此外，Kubernetes 支持命名空间功能，让不同的用户或团队在集群中看到相互隔离的不同视图。

- 水平扩展和复制

  Kubernetes 的目标是让横向扩展能够轻松进行。扩展和负载平衡都是大规模计算机中最基本的概念。

- 微服务友好

  Kubernetes 集群是采用微服务架构团队的一个完美伴侣。应用程序可以被分解成更易于开发、扩展和推导的更小单位。Kubernetes 提供了服务发现以及与其他服务进行通信的方式。

- 简化运维

  Kubernetes 可以由专门的团队来负责运维。对 Kubernetes 集群和其所运行节点的管理可以由专门的团队来负责或外包给云服务。指定应用程序的运维团队（或者开发团队自己）可以专注于应用程序的运行，而不必去具体地管理各个节点。

5.0.2 全新的概念

Docker 非常适合在单台主机上运行容器，Kubernetes 则致力于解决传统的多节点通信和规模扩展所带来的挑战。为此，Kubernetes 提出了下面一组全新的概念。

- 集群调度

  选择一个节点来运行新容器，以优化集群的可靠性和利用率的过程。

- pod

  必须将一组容器放置到同一个节点上，像一个团队一样工作。在一个节点上将一组容器紧密地联系在一起，是使应用更易管理的一种强大的方式。

- 标签（label）

  添加到 pod 的元数据，用于对容器进行分组以进行监控和管理。

- replication controller

  用于确保系统能进行水平扩展的代理（agent），也负责保证对 pod 进行可靠的管理。

- 网络服务

  一种用于在 pod 之间以及几组 pod 之间进行通信的方式，采用了动态配置的命名和网络代理。

这些概念都是什么意思呢？让我们现在就开始深入理解并使用 Kubernetes！

5.1 理解Kubernetes架构

5.1.1 问题

你需要一个容器管理系统，它需要具备可扩展性和容错能力，并且你想学习 Kubernetes 的架构（参见图 5-1）。

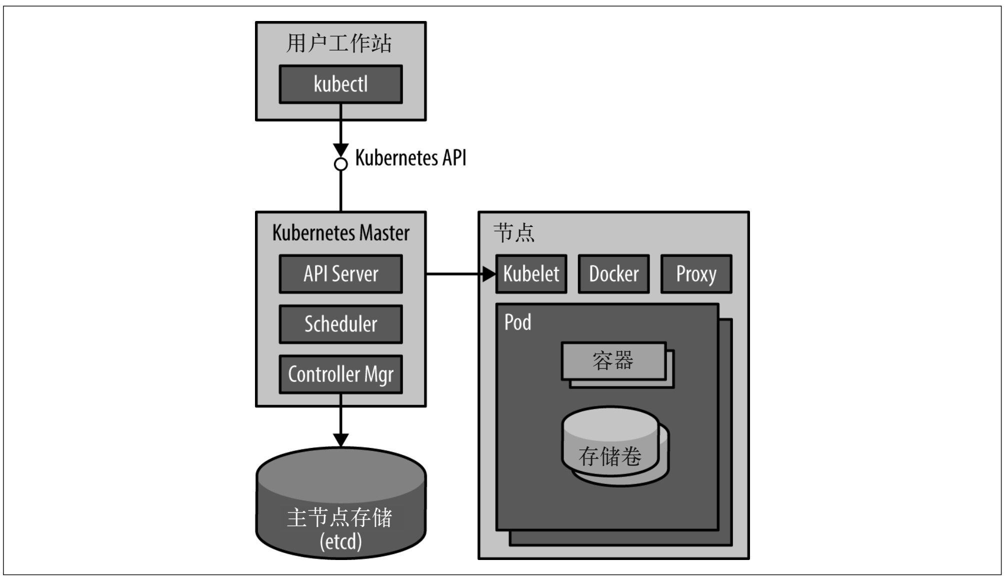


图 5-1：Kubernetes 架构图

5.1.2 解决方案

Kubernetes 的主要架构包括下面这些内容。

- Kubernetes master 服务

  这些集中式服务（可以在 Docker 容器中运行）提供了 API 来收集和展现群集的当前状态，并在节点之间分配 pod。大多数用户将始终与 master API 直接交互。这为整个集群提供了一个统一视图。

- 主节点存储

  目前，Kubernetes 所有的持久化状态都保存在 `etcd` 中。随着时间的推进，新的存储引擎会不断增加。

- kubelet

  这个代理（agent）运行在每个节点之上，负责控制 Docker，向 master 报告自己的状态，以及配置节点级别的资源（比如远程磁盘存储）。

- Kubernetes proxy

  这个代理（proxy）运行于每个节点之上（也能在其他地方运行），为本地容器提供了一个单一的网络接口，以连接到一组 pod。

5.1.3 讨论

用户通过工具（比如 kubectl）调用 Kubernetes API 和 Kubernetes master 进行交互。API 文档（由源代码自动构建）可以在 Kubernetes 的网站（http://kubernetes.io/third_party/swagger-ui/）上看到。master 节点负责存储用户想要运行的容器的描述信息（在 API 中被称为 `spec`）。之后它会将这些描述信息变为现实。它也负责报告集群的当前状态。

kubelet 和 proxy 运行在集群中的每个 worker 节点上。kubelet 负责控制 Docker 以及设置其他节点级别的状态，比如存储卷（storage volume）。proxy 负责为与服务（通常由一组在集群中运行的容器实现）进行通信提供一个稳定的本地接口。

Kubernetes 管理 pod。pod 是一组计算资源，它为一组容器提供了上下文环境。用户可以将 pod 当作一组以团队为单位进行工作的容器，pod 会被分配到同一个物理节点。虽然简单的应用只需要一个容器就能实现，但是 pod 能适用于以下各种场景。

- *多个 Docker 容器*可以共存于同一 pod 中。这非常适合一些高级场景，我们会在范例 5.7 中进行介绍。每个容器都有自己的文件系统和进程，与普通容器一样。
- pod 定义了一个*共享的网络接口*。与普通容器不一样的是，同一个 pod 中的容器都共享同一个网络接口。这样，容器之间就可以通过 `localhost` 来进行简单且高效的互相访问。这也意味着同一个 pod 中的不同容器不能使用同一个网络端口号。
- *存储卷*也是 pod 定义的一部分。如果需要，可以将这些卷映射到多个容器中。也有一些特殊类型的卷，这些卷是基于用户的需求以及集群所具备的能力。

下面是使用 Kubernetes 的基本操作流程。

(1) 通过 kubectl 工具和 Kubernetes API，用户创建一个 replication controller 规范。该规范包括一个 pod 模板以及一个希望的副本数。
(2) Kubernetes 使用这个 replication controller 中的 pod 模板来创建一组 pod。
(3) Kubernetes Scheduler（master 的一个组件）查看集群的当前状态（有哪些可用节点，以及各节点都有哪些可用资源），然后将 pod 绑定到指定节点。
(4) 该节点上的 kubelet 会观察分配给其所在节点的 pod 组中的变化，然后根据情况来启动或者终止 pod。其过程包括在需要时对存储卷进行配置，将 Docker 镜像下载到指定节点，以及通过调用 Docker API 来启动或停止各个容器。

容错则是在多个层次上实现的。

- 节点上的 kubelet 负责对 pod 内的个别容器进行健康检查和监控。
- 如果 pod 停止或者出错了，可以自动重启。

- 如果整个节点都出错了，master 节点会监测到这一状况，并且如果在等待一段时间之后发现该出错节点还没有恢复，就会删除分配到该节点上的所有 pod。这时，replication controller（如果已设置）会创建新的 pod 来取代出错节点上的旧 pod。
- 多层次的监控和重启机制能帮助应用程序在集群中出现异常（软件或硬件）时保持正常运行。

> **pod 只会被调度一次**
>
> pod 被分配到一个节点之后就不会再移动到其他节点。如果该节点丢失或者从集群中移除，这个 pod 不会被重启。这种行为有点令人惊讶，因为 Kubernetes 的目标之一就是确保能正常运行。这种措施是必需的，因为网络是不完美的。如果 master 不能连接到一个节点，则该节点上的所有 pod 都处于一种不确定的状态。就 master 而言，它不知道这些 pod 到底是处于运行状态还是未运行的状态。如果同样的 pod 在其他节点上被重新启动，就会出现两个具有同样名称和标识符的 pod 同时运行的问题。这可能会导致各种问题。例如，分布式日志可能从多个地方写入，而且都采用了相同的 pod ID 作为标识符。或者 pod ID 被用作 master 选举系统的一部分，而客户端则会对哪个 pod 才是真正的 master 感到疑惑。
>
> 实际上，为了让服务可靠地运行，我们需要使用一个 replication controller。replication controller 需要一个 pod 模板，并努力保证有指定数量的 pod 一直处于运行状态来确保 pod 能正常工作。如果发生了 master 不能联系到某一节点的情况，replication controller 会负责启动新的 pod 来代替丢失的 pod。如果通信恢复，replication controller 则会删除其中一个冗余的 pod。

5.2　用于容器间连接的网络pod

——本范例由 Joe Beda 提供

5.2.1　问题

容器在 Kubernetes 集群上调度后，你想控制网络通信如何到达容器。

5.2.2　解决方案

使用一个网络子系统来为每个容器分配一个独立的 IP 地址，这样就可以直接连接到该容器。

Kubernetes 自带了一些脚本，让你能轻松地部署到各种云服务中。很多这样的集群部署系统会自动为你进行正确的网络设置。但是，如果你正要深入研究一下它的细节，作为入门，来自 CoreOS（参见范例 6.1）的 Flannel（https://github.com/coreos/flannel）会是一个比较好的选择。

其他可选方案包括以下这些。

- 在你使用的云上构建内部网络的路由。Kubernetes 已内置对 GCE 和 Amazon EC2 的支持。

- 使用 Project Calico（http://www.projectcalico.org/）进行大规模裸机部署。
- Weave（http://weave.works/）是一种支持在大范围内进行加密通信的解决方案（参见范例 3.11）。

该问题的解决方案是使用 Kubernetes service。Kubernetes service 可以用于集群内容器之间的通信，也可以用于将外部流量转发到一组 pod。

5.2.3 讨论

在 Kubernetes 假定的网络模型中，每个 pod 都会获得一个 IP 地址。然后每个 pod 都可以连接到其他 pod，不管这些 pod 运行在哪个物理节点之上。

但是，pod 之间能够直连并不意味着这是 pod 之间最简单或者最好的通信方式。一旦一个 pod 失败或者被新的 pod 替换，调用代码必须知道如何去重连一个新的地址。这种动态重连方式非常难以集成到现有的很多服务器或者框架中。Kubernetes service 正是这一问题的解决方案（参见范例 5.8）。

5.2.4 参考

- Kubernetes 网络管理指南（http://kubernetes.io/v1.0/docs/admin/networking.html）

5.3 使用Vagrant创建一个多节点的Kubernetes集群

5.3.1 问题

你想开始用 Kubernetes，并希望在本地主机上通过 Vagrant 创建一个小型 Kubernetes 集群。

5.3.2 解决方案

你需要 Vagrant（https://vagrantup.com）和 VirtualBox（http://virtualbox.org）。如果还没有安装，请先安装这两个软件。然后设置两个环境变量：`KUBERNETES_PROVIDER`（该变量表明你将要使用 Vagrant）和 `NUM_MINIONS` [该变量用来设置集群中要启动的节点的数量（除 master 节点之外的节点数量）]。

之后你需要使用由 Kubernetes 社区提供的安装脚本（https://get.k8s.io）。这个脚本会读取环境变量，检查你的操作系统，下载最新的稳定版 Kubernetes，并解压缩到 kubernetes 目录下。下面这些命令就是在命令行上的操作步骤。

```
export KUBERNETES_PROVIDER=vagrant
export NUM_MINIONS=2
curl -sS https://get.k8s.io | bash
```

如果你没有设置 NUM_MINIONS 环境变量，那么除了 master 节点之外，只有一个节点会启动。

每个由 Vagrant 启动的虚拟机都需要 1 GB 的内存，所以请确保你拥有足够的内存。

将要下载的 Vagrant 镜像大约 316 MB，初始化虚拟机使用了 SaltStack（http://saltstack.com），这都需要一些时间。上面的操作完成之后，这些节点还会经历一个验证过程，之后你就可以在标准输出中看到类似下面的结果了。

```
...
Kubernetes cluster is running.  The master is running at:

  https://10.245.1.2

The user name and password to use is located in ~/.kubernetes_vagrant_auth.

... calling validate-cluster
Found 2 nodes.
        NAME         LABELS                             STATUS
     1  10.245.1.3   kubernetes.io/hostname=10.245.1.3   Ready
     2  10.245.1.4   kubernetes.io/hostname=10.245.1.4   Ready
Validate output:
NAME                 STATUS    MESSAGE              ERROR
etcd-0               Healthy   {"health": "true"}   nil
controller-manager   Healthy   ok                   nil
scheduler            Healthy   ok                   nil
Cluster validation succeeded
Done, listing cluster services:

Kubernetes master is running at https://10.245.1.2
KubeDNS is running at https://10.245.1.2/api/v1/proxy/namespaces/kube-system/ \
services/kube-dns
KubeUI is running at https://10.245.1.2/api/v1/proxy/namespaces/kube-system/ \
services/kube-ui
```

vagrant status 命令可以列出运行中的虚拟机，如下所示。

```
$ vagrant status
Current machine states:

master                    running (virtualbox)
minion-1                  running (virtualbox)
minion-2                  running (virtualbox)
```

到这里，你就在本地虚拟机中创建了一个可以正常工作的 Kubernetes 集群了。

5.3.3 讨论

Vagrant 使用基于 Fedora 21 和 `systemd` 的镜像创建了该集群。如果你登录到这些虚拟机，你可以列出正在运行中的 `systemd` 单元，Kubernetes 系统也正是由这些 `systemd` 单元构成的。容器之间的网络是由 Open vSwitch 创建的一个隧道。

在 master 节点上，你会看到有两个服务在运行：Addon 对象管理器和 kubelet。而 kubelet 又通过 Docker 启动了其他 Kubernetes 服务器进程，包括一个 `etcd` 实例、API server、controller manager 和 scheduler。

```
workstation$ vagrant ssh master
Last login: Tue Aug  4 23:53:35 2015 from 10.0.2.2
[vagrant@kubernetes-master ~]$ sudo systemctl list-units | grep kube
kube-addons.service    loaded active running   Kubernetes Addon Object Manager
kubelet.service        loaded active running   Kubernetes Kubelet Server
[vagrant@kubernetes-master ~]$ sudo docker ps | grep -e 'k8s_kube\|k8s_etcd' | \
awk '{print $1 " " $2}'
23963ff9ed00 gcr.io/google_containers/etcd:2.0.12
be59784f7885 gcr.io/google_containers/kube-apiserver:f8f32e739d4797f77dc3f85c...
ab3bea447298 gcr.io/google_containers/kube-scheduler:2c6e421dc8d78201f68d4cfa...
f41749ff028d gcr.io/google_containers/kube-controller-manager:4d46d90bb861fdd...
```

在 minion 节点上，你会发现两个与 Kubernetes 相关的服务：Kube-Proxy 服务和 Kubelet 服务。当然，Docker 也在运行中，如下所示。

```
workstation$ vagrant ssh minion-1
Last login: Tue Aug  4 23:52:47 2015 from 10.0.2.2
[vagrant@kubernetes-minion-1 ~]$ sudo systemctl list-units kube*
UNIT               LOAD   ACTIVE SUB     DESCRIPTION
kube-proxy.service loaded active running Kubernetes Kube-Proxy Server
kubelet.service    loaded active running Kubernetes Kubelet Server
```

使用本机上的 kubectl.sh 脚本与集群进行交互。你可以使用这个脚本对 Kubernetes 上的所有资源进行管理，从而完成容器调度任务。下面是一段 `kubectl` 的帮助信息。

```
workstation$ ./cluster/kubectl.sh
kubectl controls the Kubernetes cluster manager.

Find more information at https://github.com/GoogleCloudPlatform/kubernetes.

Usage:
  kubectl [flags]
  kubectl [command]

Available Commands:
  get            Display one or many resources
  describe       Show details of a specific resource or group of resources
  create         Create a resource by filename or stdin
  replace        Replace a resource by filename or stdin.
  patch          Update field(s) of a resource by stdin.
  delete         Delete a resource by filename, stdin, resource and name,
                 or by resources and label selector.
  namespace      SUPERCEDED: Set and view the current Kubernetes namespace
```

```
  logs           Print the logs for a container in a pod.
  rolling-update Perform a rolling update of the given ReplicationController.
  scale          Set a new size for a Replication Controller.
  exec           Execute a command in a container.
  port-forward   Forward one or more local ports to a pod.
  proxy          Run a proxy to the Kubernetes API server
  run            Run a particular image on the cluster.
  stop           Gracefully shut down a resource by name or filename.
  expose         Take a replicated application and expose it as Kubernetes
                 Service
  label          Update the labels on a resource
  config         config modifies kubeconfig files
  cluster-info   Display cluster info
  api-versions   Print available API versions.
  version        Print the client and server version information.
  help           Help about any command
...
```

为了测试是否能够与在 master 节点上运行的 Kubernetes API server 进行正常通信，你可以尝试列出集群中的节点，如下所示。

```
workstation$ ./cluster/kubectl.sh get nodes
NAME         LABELS                              STATUS
10.245.1.3   kubernetes.io/hostname=10.245.1.3   Ready
10.245.1.4   kubernetes.io/hostname=10.245.1.4   Ready
```

可以通过 ./cluster/kube-down.sh 脚本来销毁所有的虚拟机实例。

现在你就可以转到范例 5.4，通过 Kubernetes 创建你的第一个容器了。

5.3.4　参考

- Vagrant 配置文档（http://kubernetes.io/v1.0/docs/getting-started-guides/vagrant.html）
- 用于自动创建最新稳定版 Kubernetes 集群的 bash 脚本（https://get.k8s.io）

5.4　在Kubernetes集群上通过pod启动容器

5.4.1　问题

你已经知道如何使用 Docker 的命令行工具来启动容器。现在，你想使用 Kubernetes 在集群中进行容器调度。

5.4.2　解决方案

你已经拥有一个可以正常工作的 Kubernetes 集群，不管这个集群是按照范例 5.3 还是范例

5.9 创建的，或者是在公有云提供商上运行的，比如 Google 容器引擎之上。并且你已经下载了 Kubernetes 客户端 `kubectl`，设置了集群正确的 API 地址以及相应的身份验证信息（参见范例 5.15）。

范例 5.1 已经介绍过，容器是以 pod 为单位进行调度的。因此，要想启动你的第一个容器，你需要创建一个 JSON 或者 YAML 格式的 pod 定义文件，然后通过 `kubectl` 客户端将这个文件提交到 Kubernetes API server。

让我们以运行 2048 这个有趣的游戏为例来进行说明。这个游戏的 Docker 镜像可以在 Docker Hub（https://registry.hub.docker.com/u/cpk1224/docker-2048/）上找到，你也可以看到这个镜像的 Dockerfile 内容。将下面的 YAML 文件保存为 2048.yaml。

```
apiVersion: v1
kind: Pod
metadata:
  name: "2048"
spec:
  containers:
  - image: cpk1224/docker-2048
    name: "2048"
    ports:
    - containerPort: 80
      hostPort: 80
```

之后就可以将它提交到集群。

```
$ kubectl create -f 2048.yaml
pods/2048
```

当镜像下载完成之后，容器就会开始运行。你现在就可以在浏览器上打开容器所在主机的 IP 地址，玩 2048 这个游戏了。如果有防火墙，别忘了在防火墙上设置一条能访问该游戏网址的规则。

5.4.3 讨论

pod 的 YAML 配置文件的头部为 API 版本（比如 `v1`）以及资源类型（比如 `pod`）。然后需要设置一些元信息来指定 pod 名称。本例中只启动了一个容器，实际上你也可以启动多个容器。所有的容器信息都需要在 `spec` 下面的 `container` 部分进行定义。容器使用的镜像以及容器的名称都是必填信息。在这个例子中，你也定义了一个需要暴露到外部的 80 端口，并且将它映射到了宿主机的 80 端口上（通过 `containerPort` 和 `hostPort` 选项）。

可以通过 `kubectl get pods` 命令来列出集群中正在运行中的 pod。这里你会看到该 pod 将会进入运行状态，看到 pod 中有一个容器，并会看到该容器的镜像和状态，如下所示。

```
$ kubectl get pods
POD       IP           CONTAINER(S)   IMAGE(S)       HOST ...

podname   10.132.1.9                                 k8s-node/1.2.3.4 ...
                       2048           cpk1224/docker-2048 ...
```

你可以查询 pod 的定义，通过返回的 YAML 或者 JSON 格式的 pod 定义来学习 API 规范：

```
$ ./kubectl get pods -o yaml 2048
apiVersion: v1
kind: Pod
metadata:
  name: "2048"
...<snip>
```

这个 pod 通过 YAML 定义文件的 `hostPort` 属性将自己的 80 端口绑定到了宿主机的 80 端口上。在调试环境下这非常方便，但在生产环境下则不推荐这么做，因为一个宿主机的端口只能绑定一个容器。为了更灵活地将 pod 的端口暴露给外部，请使用范例 5.2 中介绍的服务。

如果已经完成了测试，就可以轻松地删除这个 pod 了，如下所示。

```
$ kubectl delete pods podname
```

5.5 利用标签查询Kubernetes对象

5.5.1 问题

在一个大规模 Kubernetes 集群中，你可能运行着数千个 pod 以及其他集群对象。你想通过标签机制在多个维度上对这些对象集合进行查询和处理。

5.5.2 解决方案

通过使用标签（label）给你的对象（比如 pod）打上标签。标签是可以添加到任何 Kubernetes 对象的键 / 值对。这些标签主要在对象描述文件的元数据部分中进行定义。

拿范例 5.4 中的例子来说，可以修改 pod 的 `yaml` 元数据描述部分，增加一个 `foo=bar` 的标签，如下所示。

```
apiVersion: v1
kind: Pod
metadata:
  name: "2048"
  labels:
    foo: bar
    version: "47"
spec:
  containers:
  - image: cpk1224/docker-2048
    name: "2048"
    ports:
    - containerPort: 80
      hostPort: 80
```

现在删除已有的名为 `2048` 的 pod，然后使用上面的新定义文件重新创建 pod。

进行上述操作后，你可以通过 `kubectl` 命令的 `--selector` 参数列出具有指定标签的所有 pod。

```
$ kubectl get pods --selector="foo=bar"
```

此外，你也可以通过 `kubectl label` 命令在运行时进行打标签操作。

```
$ kubectl label pods 2048 env=production
POD       IP          CONTAINER(S)   IMAGE(S)              HOST ...
2048      10.244.0.6                                       k8s-node/1.2.3.4 ...
                      2048           cpk1224/docker-2048
```

标签需要遵循它的规范语法（https://github.com/GoogleCloudPlatform/kubernetes/blob/master/docs/labels.md）。

简而言之，标签是一个简单的标记系统，允许用户在集群中为任何资源添加元数据。在应用生命周期的各个阶段，它有助于建立跨功能的关系来管理资源集。

现在，通过指定标签选择器来删除你的 pod。

```
$ kubectl.sh delete pod --selector="foo=bar"
pods/2048
```

5.5.3 参考

- 标签的简介、优点和语法（http://kubernetes.io/v1.0/docs/user-guide/labels.html）

5.6 使用replication controller管理pod的副本数

5.6.1 问题

你需要确保在任何时候集群中都有指定数量的 pod 副本在运行。

5.6.2 解决方案

Kubernetes 是一个声明式系统：用户定义希望这个系统去完成什么工作，而不是告诉系统如何去做。使用 replication controller，你可以为 pod 指定希望的副本数。通过在应用程序的一部分中使用服务代理（参见范例 5.8），这有助于提高系统的负荷强度和可用性。

replication controller 是 Kubernetes 集群中三大核心对象之一（其余两个为 pod 和 service）。你可以通过 `kubectl` 命令列出所有运行中的 replication controller，如下所示。

```
$ kubectl get replicationcontrollers
...
$ kubectl get rc
```

要想创建一个 replication controller，需要先编写 JSON 或者 YAML 格式的 replication controller API 规范文件。这个规范文件包含元数据信息、你希望的副本个数、一个用于标识 pod 的选择器和一个 pod 模板。目前 pod 模板嵌入在 replication controller 的定义文件中，但在将来的 Kubernetes 版本中，这可能会发生变化。

比如，你想要为范例 5.4 中的在单个 pod 中运行的 2048 游戏创建一个 replication controller，可以编写如下的 rc2048.yaml 规范文件。

```
apiVersion: v1
kind: ReplicationController
metadata:
  labels:
    name: rcgame
  name: rcgame
spec:
  replicas: 1
  selector:
    name: game
  template:
    metadata:
      labels:
        name: game
    spec:
      containers:
      - image: cpk1224/docker-2048
        name: test
        ports:
        - containerPort: 80
```

replication controller 有一个 `name=rcgame` 的标签，而 pod 则有一个 `name=game` 的标签。启动之后，replication controller 会确保任何时间都会有一个 pod 处于运行状态。你可以通过 `kubectl create` 命令创建这个 replication controller，如下所示。

```
$ kubectl create -f rc2048.yml
replicationcontrollers/rcgame
$ kubectl get rc
CONTROLLER   CONTAINER(S)  IMAGE(S)                SELECTOR         REPLICAS
rcgame       test          cpk1224/docker-2048     name=game        1
```

你可以试着终止上面创建的 pod，然后会发现，一个新的 pod 又自动创建了。

> 并不需要在启动 replication controller 之前先启动 pod。如果 pod 不存在，replication controller 会自动启动一个能匹配到定义文件中所设定标签的 pod。你也可以将副本数设置为 0。

魔法出现在当你想增加副本数的时候。你可以使用 `kubectl resize` 来进行此操作，相应的 pod 个数也会自动进行调整，如下所示。

```
$ kubectl resize --replicas=4 rc rcgame
resized
```

5.6.3　讨论

在范例 5.1 中我们提到过，Kubernetes 集群中的每个节点上都运行着一个 kubelet。这个进程监视被调度到该节点上的 pod，并确保 pod 持续运行。但是如果节点宕机了，会怎么样呢？ Kubernetes 需要一种机制来自动将该 pod 重新调度到其他节点上，并保证有足够的副本数以实现高可用性。这正是 replication controller 的工作内容。

replication controller 不仅非常有助于保证应用的可用性和弹性，在遇到诸如金丝雀部署之类的应用程序部署场景时，也是一个非常好的选择。实际上，Kubernetes 内置了基于 replication controller 的滚动升级机制，这也很值得研究，如下所示。

```
$ ./kubectl rollingupdate -h
Perform a rolling update of the given ReplicationController.

Replaces the specified controller with new controller, updating one pod at a time
to use the new PodTemplate. The new-controller.json must specify the same
namespace as the existing controller and overwrite at least one (common) label
in its replicaSelector.
...<snip>
```

5.6.4　参考

- replication controller 文档（http://kubernetes.io/v1.0/docs/user-guide/replication-controller.html）

5.7　在一个pod中运行多个容器

5.7.1　问题

你已经掌握了如何在一个 pod 中运行一个容器，但是你想在同一个 pod 中运行多个容器。你可能已经在生产环境中使用过容器，通过 Docker 链接机制来链接同一台主机上的容器。现在你希望在 Kubernetes 中也完成同样的工作。

5.7.2　解决方案

一个 pod 定义并不限于使用一个容器。你可以根据需求定义任意数量的容器和卷。在范例 5.4 中我们创建了一个 pod 定义，里面只有一个容器。下面的例子将使用 Docker Hub 上的 WordPress 和 MySQL 官方镜像启动一个 WordPress 服务。这两个镜像都以独立的容器运行，在安装时通过环境变量进行配置。WordPress 容器通过 `WORDPRESS_DB_HOST` 环境变量定义了要使用的数据库主机，并将其设置为 127.0.0.1。这将允许 WordPress 连接到同一 pod 内的 MySQL 数据库容器。这之所以能正常工作，是因为 pod 会从当前的 Kubernetes 网络模型中获取唯一的 IP 地址（参见范例 5.2）。创建如下的 wordpress.yaml 文件。

```
apiVersion: v1
kind: Pod
metadata:
  labels:
```

```
      name: wp
    name: wp
  spec:
    containers:
    - name: wordpress
      env:
      - name: WORDPRESS_DB_NAME
        value: wordpress
      - name: WORDPRESS_DB_USER
        value: wordpress
      - name: WORDPRESS_DB_PASSWORD
        value: wordpresspwd
      - name: WORDPRESS_DB_HOST
        value: 127.0.0.1
      image: wordpress
      ports:
      - containerPort: 80
        hostPort: 80
    - name: mysql
      env:
      - name: MYSQL_ROOT_PASSWORD
        value: wordpressdocker
      - name: MYSQL_DATABASE
        value: wordpress
      - name: MYSQL_USER
        value: wordpress
      - name: MYSQL_PASSWORD
        value: wordpresspwd
      image: mysql
      ports:
      - containerPort: 3306
```

创建这个 pod，如下所示。

```
$ kubectl create -f wordpress.yaml
```

当这些容器启动后，你将得到一个安装好的 WordPress。

> 可以使用 `kubectl` 命令查看该 pod 中容器的日志：
>
> ```
> $ kubectl logs wp wordpress
> ```
>
> 这里，`wp` 是启动的 pod 名称，`wordpress` 是你想要查看日志的容器的名称。

5.7.3 讨论

虽然在一个 pod 中启动多个容器很简单，但是，要想访问在这个 pod 中运行的应用，你需要使用 Kubernetes service（http://kubernetes.io/v1.0/docs/user-guide/services.html）。每个 pod 都有一个私有网络的 IP 地址。要想从 Kubernetes 集群外部通过公有网络 IP 来访问一个应用程序，需要创建一个 service 并将该应用绑定到一个公网 IP 地址，或者使用外部的负载平衡服务。

Google 容器引擎支持在描述 service 的 YAML 定义文件中直接使用一个外部负载平衡。比如，如果想让外部网络访问在本范例 pod 中运行的 WordPress 应用，你需要创建一个下面这样的 sgoogle.yml 文件。

```
apiVersion: v1
kind: Service
metadata:
  labels:
    name: wordpress
  name: wordpress
spec:
  createExternalLoadBalancer: true
  ports:
    - port: 80
  selector:
    name: wp
```

service 也有自己的元数据信息，但是更重要的是 `spec` 部分的选择器属性。在前面的例子中，`wp` 选择器会让 service 创建一个 proxy，并将负载平衡的 IP 地址绑定到能匹配到 `wp` 标签的 pod 上。如果知道负载平衡服务的 IP 地址，你就可以从外部网络访问这个服务。Kubernetes service 会将访问请求代理到该 pod 所运行的节点上。如果这个 pod 是通过 replication controller 启动的，那么 service 还会在所有运行的 pod 之间对访问请求进行负载平衡。

如果负载平衡系统还没有得到 Kubernetes 的支持，那么也可以在 service 定义文件中，手动将 pod 绑定到一个公网 IP 地址，如下所示。（这里的 `1.2.3.4` 需要替换为实际的公网 IP 地址。）

```
apiVersion: v1
kind: Service
metadata:
  labels:
    name: wordpress
  name: wordpress
spec:
  publicIPs: ["1.2.3.4"]
  ports:
    - port: 80
  selector:
    name: wp
```

5.8 使用集群IP服务进行动态容器链接

5.8.1 问题

你想将集群中多个主机上的容器链接到一起，而不是让多个容器在一个 pod 中运行。这也是一个原生云应用的设计模式，通过不同的 replication controller 来运行应用，应用的各层都可以独自进行扩展和运维。

5.8.2 解决方案

在范例 5.7 中，你启动了一个 WordPress pod，pod 里面有一个 WordPress 容器和一个 MySQL 容器。这两个容器在同一个主机上运行。你利用了一个 pod 只能有一个 IP 地址这一特点，将 WordPress MySQL 主机设置为 `localhost`。但是可以想象一下，你正在运行启用了复制功能的 MySQL 服务，或者 WordPress 前端。这种情况下，你的容器可能在集群中的不同主机上运行。

Kubernetes service 是一种智能代理，它会监视 pod 在集群中分配状态的变化，如果 pod 被重新调度了，则会自动更新这些 pod 的端口映射信息。

为了更好地运行这一典型的 WordPress 例子，我们可以让 MySQL 在一个 pod 或者 replication controller（要注意一下关于数据库复制和数据持久化的问题）中运行，然后通过 Kubernetes service 定义将 MySQL 服务向外部公开。

replication controller 的定义如下所示。

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: wp-mysql
spec:
  replicas: 1
  selector:
    tier: wp-mysql
  template:
    metadata:
      labels:
        tier: wp-mysql
    spec:
      containers:
      - name: mysql
        image: mysql
        ports:
        - containerPort: 3306
        env:
        - name: MYSQL_ROOT_PASSWORD
          value: wordpressdocker
        - name: MYSQL_DATABASE
          value: wordpress
        - name: MYSQL_USER
          value: wordpress
        - name: MYSQL_PASSWORD
          value: wordpresspwd
```

一个 MySQL service 可以定义为不同的 Kubernetes 对象类型。它也可以通过 API 来进行管理。这个 MySQL service 的定义如下所示。

```
kind: Service
apiVersion: v1
metadata:
  name: mysql
```

```
spec:
  selector:
    tier: wp-mysql
  ports:
    - port: 3306
```

注意一下 spec 部分的 selector 属性。这个选择器将会匹配到所有具有 tier: wp-mysql 标签的 pod。这个 service 将会创建一个新的“集群 IP”，集群中的其他 pod 都可以访问该 IP 地址。任何对该集群 IP 的访问都会被负载平衡代理到一个底层服务端点。

可以像下面这样查看这个 service 的底层服务接口的详细信息。

```
$ kubectl describe services/mysql
Name:               mysql
Namespace:          default
Labels:             <none>
Selector:           tier=wp-mysql
Type:               ClusterIP
IP:                 10.0.175.152
Port:               <unnamed>   3306/TCP
Endpoints:          10.244.1.4:3306
Session Affinity:   None
No events.
```

如果你正在运行（非常推荐）着 DNS 插件（add-on），这个集群 IP 还会分配一个逻辑名称，其他客户端都可以使用这个名称。在你的 WordPress pod 配置中，也可以使用这个逻辑名称。这样，不管该 IP 是什么，这个 pod 都能根据这个逻辑名称来找到数据库。可以在环境变量 WORDPRESS_DB_HOST 中看到这个值，如下所示。

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    tier: fe
  name: wp
spec:
  containers:
  - env:
    - name: WORDPRESS_DB_NAME
      value: wordpress
    - name: WORDPRESS_DB_USER
      value: wordpress
    - name: WORDPRESS_DB_PASSWORD
      value: wordpresspwd
    - name: WORDPRESS_DB_HOST
      value: mysql
    image: wordpress
    name: wordpress
    ports:
    - containerPort: 80
      hostPort: 80
      protocol: TCP
```

我们可以像范例 5.7 那样将 WordPress 服务发布到公网，这通过另一种类型为

`LoadBalancer` 的 service 来实现，如下所示。

```
apiVersion: v1
kind: Service
metadata:
  name: wordpress
spec:
  type: LoadBalancer
  ports:
    - port: 80
  selector:
    tier: fe
```

5.8.3 讨论

与 pod 和 replication controller 一样，service 也是 Kubernetes 中的关键对象。service 提供了基于 pod 的抽象，这也是在一个大型集群系统中出现错误时自发现和动态调整的关键组件。我们可以使用 service 为一组 pod 分配一个固定的名称，之后可以通过这个服务名称来可靠地访问这些 pod，不管这些 pod 被调度到了哪个节点上。

创建一个 service 将会为其分配一个新的独立于任何 pod 或者节点的集群 IP。这样，客户端就可以通过固定的方式访问服务，而不必关心服务的实现在哪里运行。当调用方建立访问 service 的连接时，在该节点上运行的本地 Kubernetes proxy 会处理这一请求。这个 proxy 会根据 service 定义将这个连接转发给某一 pod（通常通过一个标签选择器）。如果一个 service 后面有多个 pod 在运行，则这个 proxy 还会在这些 pod 之间进行负载平衡，参见图 5-2。

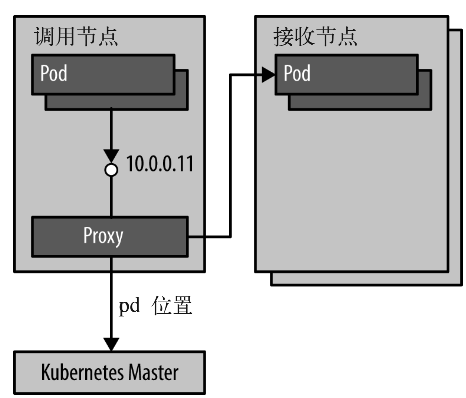


图 5-2：使用 Kubernetes 集群

调用方可以通过两种方式获得 service 的 IP 地址：环境变量或者 DNS。为 service 创建的环境变量与 Docker 容器链接中的变量很像。举例来说，你有一个名为 `redis` 的服务，该服务对外暴露了 6379 端口，则其环境变量如下所示。

```
REDIS_MASTER_SERVICE_HOST=10.0.0.11
REDIS_MASTER_SERVICE_PORT=6379
REDIS_MASTER_PORT=tcp://10.0.0.11:6379
REDIS_MASTER_PORT_6379_TCP=tcp://10.0.0.11:6379
REDIS_MASTER_PORT_6379_TCP_PROTO=tcp
REDIS_MASTER_PORT_6379_TCP_PORT=6379
REDIS_MASTER_PORT_6379_TCP_ADDR=10.0.0.11
```

这很简单，但是我们更推荐使用 DNS 来发现你的 service。在将 Kubernetes 设置为支持 DNS 时，每个 service 都会有一个可解析的名称。在这个例子中，假设默认的命名空间是 `default`，然后 DNS 的根域名被设置为 `cluster.local`，那么就可以通过 `redis.default.cluster.local` 来找到该服务。不过，如果要访问在同一个命名空间中运行的 service，也可以只使用服务名：`redis`。

5.8.4 参考

- Kubernetes 文档中的 WordPress 示例（https://github.com/kubernetes/kubernetes/blob/release-1.0/examples/mysql-wordpress-pd/README.md）
- Kubernetes service 文档（http://kubernetes.io/v1.0/docs/user-guide/services.html）

5.9 使用Docker Compose创建一个单节点Kubernetes集群

5.9.1 问题

你已经掌握了如何通过以 `systemd` 单元的方式运行集群组件（比如 API server、scheduler 和 kubelet）来创建一个 Kubernetes 集群。但是为什么不利用 Docker 本身来运行这些组件呢？如果利用的话，这将会简化集群的部署。为了测试这种部署场景，你希望在本地 Docker 容器中运行一个单节点的 Kubernetes 集群。

5.9.2 解决方案

Kubernetes 文档中有关于这种应用场景的详细说明（http://kubernetes.io/v1.0/docs/getting-started-guides/docker.html）。本范例将会更进一步，利用 Docker Compose（参见范例 7.1）来完成此工作。在开始之前，你需要一台 Docker 主机，并且要事先安装好 Docker Compose。你可以克隆随本书附带的 Git 仓库，使用里面提供的 Vagrantfile。

```
$ git clone https://github.com/how2dock/docbook.git
$ cd docbook/ch05/docker
$ tree
.
├── Vagrantfile
├── k8s.yml
└── kubectl
```

这个 Vagrantfile 文件有一段初始化脚本，这个脚本会在虚拟机中安装 Docker 和 Docker Compose。k8s.yml 是 `docker-compose` 的配置文件，定义了用于以容器的方式启动 Kubernetes 所需要的全部组件。启动这个虚拟机，然后运行 `docker-compose` 命令，就会下载所有需要的镜像，并启动这些容器，如下所示。

```
$ vagrant up
$ vagrant ssh
$ cd /vagrant
$ docker-compose -f k8s.yml up -d
$ docker ps
CONTAINER ID   IMAGE                          COMMAND
64e0073615c5   gcr.io/google_containers/...   "/hyperkube controll  ...
9603f3b5b186   gcr.io/google_containers/...   "/hyperkube schedule  ...
3ce44e77989f   gcr.io/google_containers/...   "/hyperkube apiserve  ...
1b0bcbb56d59   kubernetes/pause:go            "/pause"              ...
0b0c3e2735a9   kubernetes/etcd:2.0.5.1        "/usr/local/bin/etcd  ...
459c45ef9389   gcr.io/google_containers/...   "/hyperkube proxy --  ...
005c5ac1de0e   gcr.io/google_containers/...   "/hyperkube kubelet   ...
```

这就是全部的工作了。现在你已经有了一个单节点的 Kubernetes 集群，它所有的组件都在容器中运行。`get nodes` 命令会返回你的 `localhost`，你也可以创建 pod、replication controller 和 service，如下所示。

```
$ ./kubectl get nodes
NAME        LABELS    STATUS
127.0.0.1   <none>    Ready
```

可以通过创建一个 Nginx 容器，来测试一下你是否能够创建新 pod，如下所示。

```
$ ./kubectl run-container nginx --image=nginx --port=80
CONTROLLER   CONTAINER(S)   IMAGE(S)   SELECTOR              REPLICAS
nginx        nginx          nginx      run-container=nginx   1
```

> `run-container` 命令会自动为要启动的容器创建一个 replication controller。你可以通过 `./kubectl get rc` 命令来查看 replication controller 列表。

要想从集群外部访问 nginx 前台服务，需要将 nginx 暴露为一个 service。但是，在创建 service 时，只需要将虚拟机的仅主机网络 IP 地址传给 `kubectl` 命令。否则，创建该服务之后，虽然新启动的 pod 能访问到该服务，但是我们将无法从群集外部来访问它，如下所示。

```
$ ./kubectl expose rc nginx --port=80 --public-ip=192.168.33.10
NAME      LABELS    SELECTOR              IP          PORT
nginx     <none>    run-container=nginx   10.0.0.98   80
```

在 nginx 镜像下载完成之后，你就能通过 http://192.168.33.10 来访问 pod 中的 Nginx 欢迎页面了，如下所示。

```
$ ./kubectl get pods
POD          IP           CONTAINER(S)         IMAGE(S)                       ...
nginx-127                 controller-manager   gcr.io/google_containers/...
                          apiserver            gcr.io/google_containers/...
                          scheduler            gcr.io/google_containers/...
nginx-461yi  172.17.0.6   nginx                nginx                          ...
```

5.9.3 讨论

Docker Compose 的配置文件 k8s.yml 显示了这一切都是如何进行的。

```
etcd:
  image: kubernetes/etcd:2.0.5.1
  net: "host"
  command: /usr/local/bin/etcd --addr=127.0.0.1:4001 --bind-addr=0.0.0.0:4001 \
  --data-dir=/var/etcd/data
master:
  image: gcr.io/google_containers/hyperkube:v0.14.1
  net: "host"
  volumes:
    - /var/run/docker.sock:/var/run/docker.sock
  command: /hyperkube kubelet --api_servers=http://localhost:8080 --v=2 \
  --address=0.0.0.0 \
           --enable_server --hostname_override=127.0.0.1 \
           --config=/etc/kubernetes/manifests
proxy:
  image: gcr.io/google_containers/hyperkube:v0.14.1
  net: "host"
  privileged: true
  command: /hyperkube proxy --master=http://127.0.0.1:8080 --v=2
```

Compose 启动了三个容器：一个容器运行 etcd，一个容器运行 Kubernetes 代理服务，还有一个容器运行 Kubernetes kubelet。服务代理和 kubelet 都来源于同一镜像，并且使用了同一个可执行程序，通过命令行参数来区分。这个可执行程序为 hyperkube，是一个非常好用的工具类程序，你可以用这个命令启动 Kubernetes 集群中的所有组件。

这其中的巧妙之处是，master 容器调用 hyperkube 时指定了一个位于该容器镜像中 /etc/kubernetes/manifests 文件夹下面的配置文件。你可以启动一个临时容器查看一下该清单文件的内容，如下所示。

```
$ docker run --rm -it gcr.io/google_containers/hyperkube:v0.14.1 cat /etc/ \
kubernetes/manifests/master.json
{
"apiVersion": "v1beta3",
"kind": "Pod",
"metadata": {"name":"nginx"},
"spec":{
  "hostNetwork": true,
  "containers":[
    {
      "name": "controller-manager",
      "image": "gcr.io/google_containers/hyperkube:v0.14.1",
      "command": [
```

```
                "/hyperkube",
                "controller-manager",
                "--master=127.0.0.1:8080",
                "--machines=127.0.0.1",
                "--sync_nodes=true",
                "--v=2"
          ]
        },
        {
          "name": "apiserver",
          "image": "gcr.io/google_containers/hyperkube:v0.14.1",
          "command": [
                "/hyperkube",
                "apiserver",
                "--portal_net=10.0.0.1/24",
                "--address=127.0.0.1",
                "--etcd_servers=http://127.0.0.1:4001",
                "--cluster_name=kubernetes",
                "--v=2"
          ]
        },
        {
          "name": "scheduler",
          "image": "gcr.io/google_containers/hyperkube:v0.14.1",
          "command": [
                "/hyperkube",
                "scheduler",
                "--master=127.0.0.1:8080",
                "--v=2"
          ]
        }
      ]
    }
  }
```

这个清单文件会发送给 kubelet，kubelet 会启动该文件所定义的容器。在这个例子中，它会启动 Kubernetes 的 API server、scheduler 和 controller manager。这三个组件组成了一个 Kubernetes pod，它们自己本身也会由 kubelet 负责监控。实际上，如果你查询正在运行中的 pod，将会得到如下结果。

```
$ ./kubectl get pods
POD        IP   CONTAINER(S)         IMAGE(S)
nginx-127       controller-manager   gcr.io/google_containers/hyperkube:v0.14.1
                apiserver            gcr.io/google_containers/hyperkube:v0.14.1
                scheduler            gcr.io/google_containers/hyperkube:v0.14.1
```

5.9.4　参考

- 在本地通过 Docker 运行 Kubernetes（http://kubernetes.io/v1.0/docs/getting-started-guides/docker.html）

5.10 编译Kubernetes构建自己的发布版本

5.10.1 问题

你希望从源代码构建 Kubernetes 可执行程序，而不是下载官方发布的版本。

5.10.2 解决方案

Kubernetes 采用 Go 语言编写，它的构建系统使用了 Docker，所有的构建都在容器中进行。你也可以不使用容器，而是使用本地的 Go 环境来构建 Kubernetes，但是采用容器构建能极大地简化环境搭建。因此，要想构建 Kubernetes 程序，你需要安装 Go 语言软件包、Docker 以及 Git 来从 GitHub 下载源代码。比如在 Ubuntu 14.04 系统上，如下所示。

```
$ sudo apt-get update
$ sudo apt-get -y install golang
$ sudo apt-get -y install git
$ sudo curl -sSL https://get.docker.com/ | sudo sh
```

确认你的 Go 和 Docker 已经成功安装，如下所示。

```
$ go version
go version go1.2.1 linux/amd64
$ docker version
Client version: 1.6.1
Client API version: 1.18
Go version (client): go1.4.2
Git commit (client): 97cd073
OS/Arch (client): linux/amd64
Server version: 1.6.1
Server API version: 1.18
Go version (server): go1.4.2
Git commit (server): 97cd073
OS/Arch (server): linux/amd64
```

克隆 Kubernetes 的 Git 仓库以获得 Go 源代码，如下所示。

```
$ git clone https://github.com/GoogleCloudPlatform/kubernetes.git
$ cd kubernetes
```

现在就可以进行构建了。在 /build 文件夹下有一个名为 run.sh 的构建脚本，直接使用这个文件即可。脚本执行时会询问是否要下载 Golang 的 Docker 镜像，然后开始构建。以下构建过程的一部分输出片段。

```
$ ./build/run.sh hack/build-go.sh
+++ [0513 11:51:46] Verifying Prerequisites....
You don't have a local copy of the golang docker image. This image is 450MB.
Download it now? [y/n] Y
...<snip>
+++ [0513 11:58:08] Placing binaries
+++ [0513 11:58:14] Running build command....
+++ [0513 11:58:16] Output directory is local.  No need to copy results out.
```

构建出来的可执行文件会保存在 _output 文件夹下。如果你是在 64 位的 Linux 主机上进行的构建，那么构建结果会保存在 _output/dockerized/bin/linux/amd64 下面，如下所示。

```
~/kubernetes/_output/dockerized/bin/linux/amd64# tree
.
├── e2e
├── genbashcomp
├── gendocs
├── genman
├── ginkgo
├── hyperkube
├── integration
├── kube-apiserver
├── kube-controller-manager
├── kubectl
├── kubelet
├── kube-proxy
├── kubernetes
├── kube-scheduler
└── web-server
```

5.10.3 讨论

类似地，你也可以完整地构建一个发布制品，这些制品以一个压缩包 kubernetes.tar.gz 的形式交付。压缩包里面包含了所有的可执行程序、示例、插件和部署脚本。创建这个发布包比单纯地构建可执行程序要花费更多的时间，所有端到端的测试都会被执行一遍。为了构建一个完整的发布包，执行下面的命令，并在 /_output/release-tars/ 文件夹下面查看发布包的构建结果。

```
$ ./build/release.sh
$ tree _output/release-tars/
_output/release-tars/
├── kubernetes-client-darwin-386.tar.gz
├── kubernetes-client-darwin-amd64.tar.gz
├── kubernetes-client-linux-386.tar.gz
├── kubernetes-client-linux-amd64.tar
├── kubernetes-client-linux-arm.tar.gz
├── kubernetes-client-windows-amd64.tar.gz
├── kubernetes-salt.tar.gz
├── kubernetes-server-linux-amd64.tar.gz
├── kubernetes.tar.gz
└── kubernetes-test.tar.gz
```

除了压缩包之外，构建发布包的过程中还会为 Kubernetes 集群的三个主要组件创建三个镜像：API server、controller 和 scheduler，如下所示。

```
# docker images
REPOSITORY                                          ...
gcr.io/google_containers/kube-controller-manager    ...
gcr.io/google_containers/kube-scheduler             ...
gcr.io/google_containers/kube-apiserver             ...
```

发布包中有一个 Dockerfile 文件，由这个 Dockerfile 文件构建的镜像里面包含 hyperkube 可执行程序。这个程序可以用来启动 Kubernetes 集群中的所有组件。在范例 5.9 中，我们使用这个程序在一个单节点上通过 Docker 容器创建了一个 Kubernetes 集群。你可以使用这个 Dockerfile 去构建自己的 Hyperkube 镜像，并根据自己的需求来编辑配置文件 master.json，如下所示。

```
    $ tree kubernetes/cluster/images/hyperkube/
    kubernetes/cluster/images/hyperkube/
├── Dockerfile
├── Makefile
├── master.json
└── master-multi.json
```

5.10.4 参考

- 构建 Kubernetes 程序的 README 文件（https://github.com/kubernetes/kubernetes/blob/master/build/README.md）
- 使用 godep 的开发环境（https://github.com/kubernetes/kubernetes/blob/master/docs/devel/development.md）

5.11 使用hyperkube二进制文件启动Kubernetes组件

5.11.1 问题

一个 Kubernetes 集群由一个 master 节点和多个 worker 节点构成。每个节点都运行着若干 Kubernetes 可执行程序。为了方便开发，你希望只使用一个可执行程序，将你想要启动的组件通过命令行参数的形式传递给该命令。

5.11.2 解决方案

使用 hyperkube。

如范例 5.10 的提示部分所述，一个发布包包括了所有的 Kubernetes 组件的可执行程序：API server、controller manager、scheduler、服务代理以及 kubelet。后面两个程序会在每个 worker 节点上运行，而前三个组件再加上 `etcd` 则构成了 Kubernetes master。hyperkube 是一个可执行文件，你可以用这个命令来启动所有这些 Kubernetes 组件。

如果你已经按照范例 5.10 中介绍的那样构建了自己的发布包，你可以在 _output/ 文件夹下面找到 hyperkube 文件，如下所示。

```
# tree ~/kubernetes/_output/release-tars/kubernetes/server/kubernetes/server/bin
/root/kubernetes/_output/release-tars/kubernetes/server/kubernetes/server/bin
├── hyperkube
```

```
├── kube-apiserver
├── kube-apiserver.docker_tag
├── kube-apiserver.tar
├── kube-controller-manager
├── kube-controller-manager.docker_tag
├── kube-controller-manager.tar
├── kubectl
├── kubelet
├── kube-proxy
├── kubernetes
├── kube-scheduler
├── kube-scheduler.docker_tag
└── kube-scheduler.tar
```

使用 hyperkube 时，需要指定想要启动的组件（比如 `apiserver`、`controller-manager`、`scheduler`、`kubelet` 或者 `proxy`）。在指定组件名参数之后，你可以指定其他所有需要设置的参数。比如，要想启动 API server，你可以像下面这样，查看 hyperkube 命令的使用方法。

```
$ ./hyperkube apiserver -h
The main API entrypoint and interface to the storage system. The API server is
also the focal point for all authorization decisions.

Usage:
  apiserver [flags]

Available Flags:
      --address=127.0.0.1: DEPRECATED: see --insecure-bind-address instead
      --admission-control="AlwaysAdmit": Ordered list of plug-ins to do ...
      --admission-control-config-file="": File with admission control ...
      --allow-privileged=false: If true, allow privileged containers.
<snip>
```

5.12 浏览Kubernetes API

5.12.1 问题

Kubernetes 提供了一套 REST API，你需要学习这些 API 以便对集群进行管理，并在集群中运行应用程序。

5.12.2 解决方案

Kubernetes 提供了一个带版本号的 API。在使用 v1 版本的 API 时，用户希望针对该版本不会有不兼容的变化。API server 同时支持多个版本的 API，不过大多数用户应该使用 v1 版的 API。

如范例 5.9 中所讲到的，你可以在本地通过 Docker 启动一个 Kubernetes 集群。这是体验 Kubernetes 最简单的方式。所有组件都启动之后，你就可以访问 API server 提供的 API 了。如果你正在使用运行 API server 的计算机，可以通过 http://localhost:8080 来访问 API，而且不需要任何身份验证。你可以使用 `curl` 来完成第一次 Kubernetes 原生 API 体验。可以

通过访问 http://localhost:8080/api 来列出目前支持的所有 API 版本，如下所示。

```
$ curl http://localhost:8080/api
{
  "versions": [
    "v1",
  ]
}
```

从上面的返回结果可以看到，这个服务器只提供了 v1 版本的 API。如果你的输出中是类似 `v1beta3` 的内容，那么说明你运行的是老版本的 Kubernetes。如果你看到的结果类似 `v2beta1`，说明你正在使用的是开发体验版。为了确认你正在使用的 API 版本号，可以运行 `curl` 来访问 http://localhost:8080/version。需要注意的是，API 的版本与可执行文件的版本号不需要统一，如下所示。

```
$ curl http://localhost:8080/version
{
  "major": "1",
  "minor": "0",
  "gitVersion": "v1.0.1",
  "gitCommit": "6a5c06e3d1eb27a6310a09270e4a5fb1afa93e74",
  "gitTreeState": "clean"
}
```

上面的输出告诉你，在这个例子中，你所使用的是官方的 1.0.1 版本的 Kubernetes。这只是关于 API 最基本的信息，没有给出完整的 API 视图。不过值得庆幸的是，Kubernetes API 文档使用了 Swagger（http://swagger.io）。这就是说，我们可以通过 /swaggerapi/ 接口来得到所有可用的 API 接口列表，如下所示。

```
$ curl http://localhost:8080/swaggerapi/
{
  "swaggerVersion": "1.2",
  "apis": [
   {
    "path": "/api/v1",
    "description": "API at /api/v1 version v1"
   },
   {
    "path": "/api",
    "description": "get available API versions"
   },
   {
    "path": "/version",
    "description": "git code version from which this is built"
   }
  ],
  "apiVersion": "",
  "info": {
   "title": "",
   "description": ""
  }
 }
```

之后可以通过 `curl` 命令来获得每个 API 的详细 JSON 规范，如下所示。

```
$ curl http://localhost:8080/swaggerapi/api/v1
```

如果你想编写自己的 Kubernetes 客户端，这会非常有用。但是，Swagger 同时提供了一个 Web 界面以便于查看这些 API。假设你能从浏览器访问 API server，在浏览器中打开 http://<KUBE_MASTER_IP>:8080/swagger-ui/，就应该能看到类似图 5-3 的 Swagger 界面。

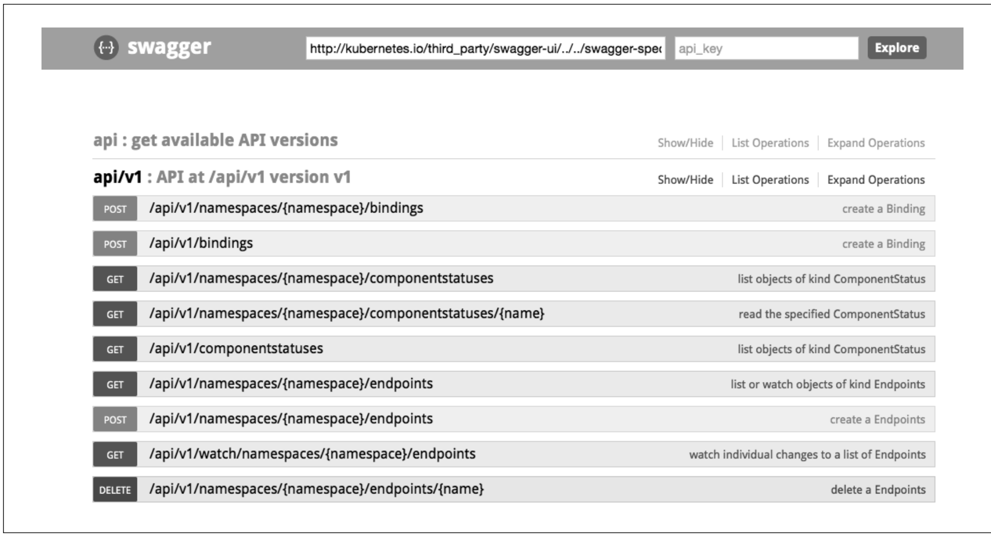


图 5-3：通过 Swagger UI 查看 Kubernetes API

5.12.3　讨论

通过 Swagger 和 `curl` 浏览 Kubernetes API 非常利于你更好地理解 Kubernetes，包括用于定义 pod、replication controller 和 service 的架构。但是更多地去尝试使用 `kubectl` 客户端更加实用，每个发布版本都会有这个程序。而且它的使用方法也都有良好的文档进行说明，这个程序也使用了很多 Kubernetes API，如下所示。

```
$ ./kubectl
kubectl controls the Kubernetes cluster manager.

Find more information at https://github.com/GoogleCloudPlatform/kubernetes.

Usage:
  kubectl [flags]
  kubectl [command]

Available Commands:
  get           Display one or many resources
  describe      Show details of a specific resource or group of resources
  create        Create a resource by filename or stdin
  replace       Replace a resource by filename or stdin.
  patch         Update field(s) of a resource by stdin.
```

```
  delete         Delete a resource by filename, stdin, resource and name, ...
  namespace      SUPERCEDED: Set and view the current Kubernetes namespace
  logs           Print the logs for a container in a pod.
  rolling-update Perform a rolling update of the given ReplicationController.
  scale          Set a new size for a Replication Controller.
  exec           Execute a command in a container.
  port-forward   Forward one or more local ports to a pod.
  proxy          Run a proxy to the Kubernetes API server
  run            Run a particular image on the cluster.
  stop           Gracefully shut down a resource by name or filename.
  expose         Take a replicated application and expose it as Kubernetes
                 Service
  label          Update the labels on a resource
  config         config modifies kubeconfig files
  cluster-info   Display cluster info
  api-versions   Print available API versions.
  version        Print the client and server version information.
  help           Help about any command

<snip>
```

在浏览 Kubernetes API 时，你可能会体验到一些比较有趣的接口，比如 /ping/ 和 /validate：

```
$ curl http://localhost:8080/ping/
{
  "paths": [
    "/api",
    "/api/v1",
    "/healthz",
    "/healthz/ping",
    "/logs/",
    "/metrics",
    "/resetMetrics",
    "/swagger-ui/",
    "/swaggerapi/",
    "/ui/",
    "/version"
  ]
}
```

5.12.4 参考

- Kubernetes API 文档（http://kubernetes.io/v1.0/docs/api.html）
- 访问 Kubernetes API（http://kubernetes.io/v1.0/docs/admin/accessing-the-api.html）
- Kubernetes API 约定（http://kubernetes.io/v1.0/docs/devel/api-conventions.html）

5.13 运行Kubernetes仪表盘

5.13.1 问题

你想对你的 Kubernetes 集群进行可视化，以深入了解正在运行的各个实体（包括 pod、service 和 replication controller）。

5.13.2 解决方案

从 Kubernetes 0.16 开始，API server 自带了一个 Web 用户界面。因此，如果你能让 API server 在一个可以通过浏览器访问到的地址上运行，就可以直接通过 `/static/app` 来访问这个 Web UI。

举例来说，虽然这种方式不是很安全，你可以通过 http://<KUBE_MASTER_IP>: 8080/static/app 来访问 Web UI。图 5-4 是一个屏幕截图。

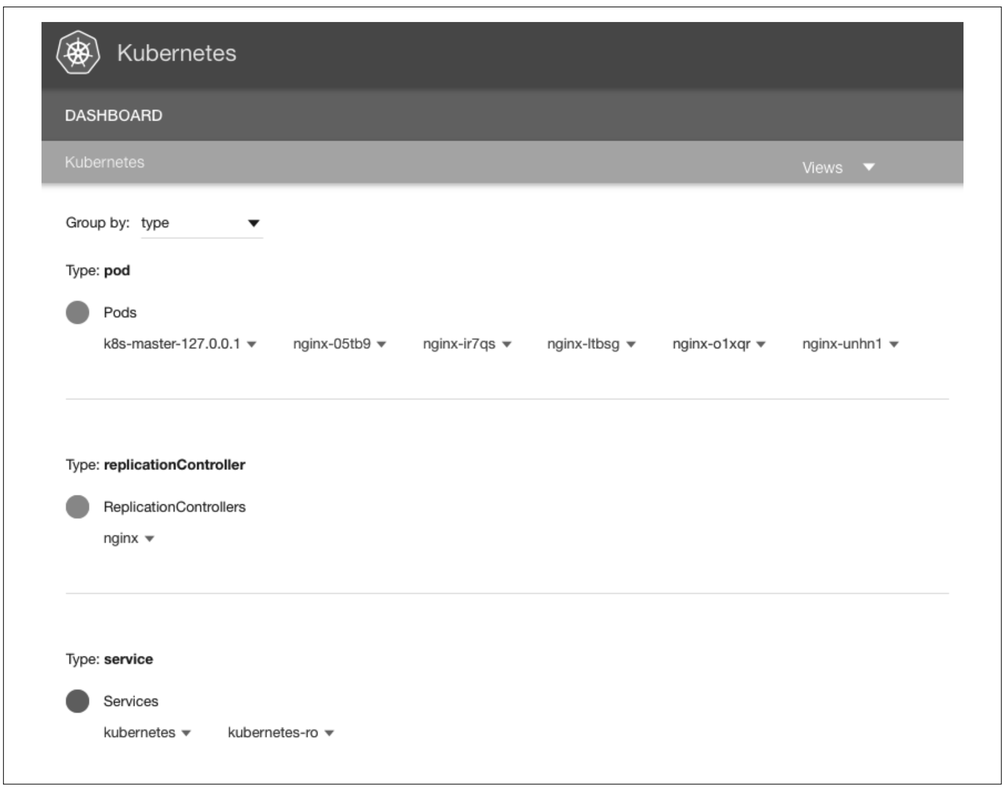


图 5-4：Kubernetes 仪表盘

现在，你只能通过这个 Web UI 来进行一些查看操作，还不能对 pod、service 或者 replication controller 进行管理。不久的将来，这也应该会有所变化。

5.13.3　讨论

现在，来自 Kismatic（https://kismatic.io）团队的成员正在对 Kubernetes 仪表盘功能进行活跃的开发，除了查看视图可能会频繁变更之外，用来通过 Web UI 对 Kubernetes 组件进行管理的新功能也会不断添加进来。

Kubernetes 仪表盘的源代码（https://github.com/GoogleCloudPlatform/kubernetes/tree/master/www）里面有关于如何配置开发环境的详细文档。你也可以编写自己的可视化组件（https://github.com/GoogleCloudPlatform/kubernetes/blob/master/www/master/components/README.md），这在 Kubernetes Web UI 中被称为组件。

5.14　升级老版本API

5.14.1　问题

你正在使用 v1 版本的 API，但是你已有的配置文件可能使用了较老的 beta 版本的 API。你需要一个工具来帮助你对所有配置文件进行升级。

5.14.2　解决方案

> 本范例只是为了辅助开发人员。它可能会被废弃，不应该在生产环境下使用。这个工具只用于 v1 版本 API 的开发，并不保证在老版本的代码中也能正常工作。

使用 `kube-version-change` 命令，这是用 Golang 编写的程序。可以在源代码的 /cmd/ 文件夹下找到这个程序。

假设你是按照范例 5.10 来操作的，那么你已经为本次测试做好了准备。如果还没有从源代码构建 Kubernetes，那么你需要现在就从源代码构建 Kubernetes（参见范例 5.10）。

在从 GitHub 下载下来的 Kubernetes 源代码的根文件夹下，执行下面的命令。

```
$ hack/build-go.sh cmd/kube-version-change
```

这将会使用你本地的 Golang 构建 `kube-version-change` 程序，构建结果将会保存在 /_output/local/bin/ 文件夹下。在 64 位的 Linux 计算机上，这个程序会被保存到 _output/dockerized/bin/linux/amd64/kube-version-change 下。

5.14.3　讨论

当版本变更工具编译完成之后，就可以将你的配置文件迁移到新版的 API 了。假设你有一个 MySQL 的 pod 定义文件（mysql.yaml），这个配置文件使用的是版本为 `v1beta2` 的 API 规范，文件内容如下所示。

```
apiVersion: v1beta2
desiredState:
  manifest:
    containers:
      - name: mysql
        image: mysql
        env:
        - name: MYSQL_ROOT_PASSWORD
          value: password
        ports:
        - containerPort: 3306
          name: mysql
          protocol: TCP
id: mysql
kind: Pod
labels:
  name: mysql
```

将版本修改为 v1。

```
$ ./kube-version-change -i mysql.yaml -o mysql3.yaml
```

这条命令会创建一个名为 mysql3.yaml 的 pod 定义文件，并使用了 `v1beta3` API 规范定义了 pod，如下所示。

```
apiVersion: v1beta3
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    name: mysql
  name: mysql
spec:
  containers:
  - capabilities: {}
    env:
    - name: MYSQL_ROOT_PASSWORD
      value: password
    image: mysql
    imagePullPolicy: IfNotPresent
    name: mysql
    ports:
    - containerPort: 3306
      name: mysql
      protocol: TCP
    resources: {}
    securityContext:
      capabilities: {}
      privileged: false
    terminationMessagePath: /dev/termination-log
  dnsPolicy: ClusterFirst
  restartPolicy: Always
  serviceAccount: ""
  volumes: null
status: {}
```

你也可以将 API 版本从 `v1beta3` 迁移到之前的老版本。这可以很方便地帮助我们理解规范。尝试下面的命令。

```
$ ./kube-version-change -i mysql3.yaml -o mysql2.yaml -v v1beta2
```

5.15 为Kubernetes集群添加身份验证支持

5.15.1 问题

你想创建一个具备身份验证和授权功能的 Kubernetes 集群。这样，用户就能通过 Kubernetes 客户端（比如 `kubectl`）以一种安全的方式对集群进行管理。

5.15.2 解决方案

启动 API server 时，指定如下选项之一：`--token_auth_file`、`--basic_auth_file` 或者 `--client_ca_file`。你也需要确保没有将 API server 绑定到一个不安全的公网 IP 地址上。

默认情况下，Kubernetes API server 会在 6443 端口上启用 HTTPS 监听，并使用一个自签名的证书。也可以通过 `--tls-cert-file` 和 `--tls-private-key-file` 选项指定自己的证书。

出于测试或者学习的目的，你可能会使用 `--insecure-bind-address=0.0.0.0` 参数启动 API server，这会以被称为 localhost port 的方式绑定到所有的网络接口上，包括 Kubernetes master 节点的公网 IP。这非常方便，你可以直接通过 http://<KUBE_MASTER_IP>:8080，不经过身份验证就能访问到你的集群，但这样做一点都不安全。

默认情况下，Kubernetes 会将 7080 端口的只读访问绑定到所有网络接口上。如果你的防火墙对外开放了 7080 端口，则整个集群都可以从外部公开访问。但是，这一机制应该在 Kubernetes v1.0 之前有所改变。

5.15.3 讨论

基本身份验证和基于令牌的身份验证所使用的文件格式都是非常简单的 CSV 文件。关于身份验证的相关文档（https://github.com/GoogleCloudPlatform/kubernetes/blob/master/docs/authentication.md）也指向了相关代码（https://github.com/kubernetes/kubernetes/tree/master/plugin/pkg/auth/authenticator）的位置。密切关注这些身份验证插件非常有用，因为这些身份验证机制有可能会废弃或者发生变化。目前，令牌过期和密码重置等功能都还没有实现。

比如，你可以创建如下的基本身份验证文件并保存到 /tmp/auth 文件夹下，文件内容遵循 `password,username,useruid` 的格式。

```
foobar,admin,1000
```

通过 hyperkube 启动 API server（参见范例 5.11），并指定如下选项。

```
$ hyperkube apiserver --portal_net=10.0.0.1/24
                      --etcd_servers=http://127.0.0.1:4001
                      --cluster_name=kubernetes
                      --basic_auth_file=/tmp/auth
                      --v=2
```

这会使用一些默认的参数设置。HTTPS 将会监听 6443 端口，只读访问将会被绑定到 7080 端口，而本地使用的端口只会被绑定到 localhost 上。如果你的防火墙没有打开 7080 端口，则你的 Kubernetes 集群将只能使用 HTTPS 进行基本身份验证。

> 基本身份验证将来会被废弃，取而代之的是基于令牌或者客户端证书的身份验证机制。目前提供基本身份验证功能只是为了方便而已。只读访问模式在将来的版本中也将会被移除。

5.15.4 参考

- 安全地访问 API server（https://github.com/GoogleCloudPlatform/kubernetes/blob/master/docs/accessing_the_api.md）
- 访问 Kubernetes 集群（https://github.com/GoogleCloudPlatform/kubernetes/blob/master/docs/accessing-the-cluster.md）
- Kubernetes 身份验证插件（https://github.com/GoogleCloudPlatform/kubernetes/blob/master/docs/authentication.md）
- Kubernetes 身份验证功能路线图（https://github.com/GoogleCloudPlatform/kubernetes/blob/master/docs/authorization.md）

5.16 配置Kubernetes客户端连接到远程集群

5.16.1 问题

你通过一种身份验证机制将 API server 安全地对外公开，希望你的用户能使用某一客户端（比如 `kubectl`）从远程访问这个集群。

5.16.2 解决方案

使用 `kubectl` 配置创建多个上下文环境来访问你的集群。在每个上下文环境中，指定集群 API 接口地址和用户身份验证信息。

实际上，默认情况下 `kubectl` 会连接到 localhost 上的 API server。但是你可以定义多个接口地址（比如在不同的地区有多个集群的时候就很有用）和多个用户属性（比如 production、development 和 service），它们可以有不同的身份验证策略。第一次安装 `kubectl` 时，这些配置都是空的。你可以通过运行 `kubectl config view` 命令来查看当前的配置，如下所示。

```
$ ./kubectl config view
apiVersion: v1
clusters: []
contexts: []
current-context: ""
kind: Config
preferences: {}
users: []
```

你可以用多个选项来定义一个集群、一个上下文环境以及一些用户身份验证信息。下面是一个例子，这里定义了一个名为 `k` 的集群，使用了 HTTPS 的接口地址，并使用了自签名的证书；之后又创建了一个上下文环境 `kcon`，它使用了集群 `k` 和用户 `superadmin`。`superadmin` 用户的身份验证信息已经在范例 5.15 中创建。在这个例子的最后，你通过 `use-context` 命令设置了当前的上下文环境。这些设置完成之后，再次使用 `kubectl` 命令时，它能够正确地拼接出 HTTP 请求，并安全地通过身份验证来访问远程的 Kubernetes 集群，如下所示。

```
$ ./kubectl config set-cluster k --server=https://<KUBE_MASTER_PUBLIC_IP>:6443 \
                                 --insecure-skip-tls-verify=true
$ ./kubectl config set-context kcon --user=superadmin
$ ./kubectl config set-context kcon —-cluster=k
$ ./kubectl config set-credentials superadmin --username=admin --password=foobar
$ ./kubectl config use-context kcon
```

5.16.3 讨论

虽然 `kubectl` 客户端功能强大，但是应该记住，你也可以编写自己的客户端，因为客户端发出的也都是标准的 HTTP 请求。举例来说，你可以通过 `curl` 来发起一个身份验证请求。

```
$ curl -k -u toto:foobar https://<KUBE_MASTER_PUBLIC_IP>:6443/api
{
  "versions": [
    "v1"
  ]
}
```

5.16.4 参考

- Kubernetes 客户端库（https://github.com/GoogleCloudPlatform/kubernetes/blob/master/docs/client-libraries.md）

第 6 章

为Docker优化的操作系统

6.0 简介

在第 1 章中，我们讨论了一些安装场景，主要展示了如何在传统的操作系统上安装 Docker。本章将讲述专门针对 Docker 进行过优化的新一代操作系统。这些新操作系统着眼于两个新的发展趋势。首先，它们认为在服务器之上运行的一切都将是一个容器。其次，它们尝试实施原子升级机制，以简化在数据中心对服务器的运维工作。这意味着这些新操作系统并不会再提供像 `yum` 或 `apt` 这样的传统软件包管理器，而是假设你会在服务器上通过下载镜像并启动一个容器的方式来运行你的应用程序。这些操作系统只提供运行容器所要满足的最低要求。

我们将要介绍的第一个操作系统是 CoreOS（参见范例 6.1）。CoreOS 已经可以在多个公有云服务上使用。CoreOS 也可以安装在裸机上，通过 Vagrant 在本地进行测试，或者建立自己的 ISO 镜像。在范例 6.2 中，我们会演示如何配置 CoreOS 实例。在范例 6.3 中，我们将介绍如何创建 CoreOS 服务器集群。在范例 6.4 中，我们会讲述如何使用基于系统原生的调度器在 CoreOS 集群中启动容器。之后，我们会介绍 Flannel，它是捆绑在 CoreOS 中的覆盖（overlay）网络技术。正如我们在第 3 章中所提到的那样，Flannel 是为跨越多台主机的容器通过一个私有 IP 空间进行通信的一个网络解决方案。

之后，我们将介绍其他三个专门针对 Docker 进行过优化的操作系统。在范例 6.6 中，我们会对 RedHat 的 Atomic 进行介绍，在范例 6.7 中会演示如何在 AWS 上启动 Atomic 实例。在范例 6.8 中，我们将会介绍 Ubuntu Snappy，而在范例 6.9 中，还会学习如何在 AWS 上启动一个 Ubuntu 实例。除了介绍的这些基于 AWS 的例子，你还可以选择在本地计算机上或是在云环境中尝试这些新一代操作系统。最后，我们会在范例 6.10 中对 RancherOS 进行说明，RancherOS 与其他操作系统的不同之处在于它的所有系统服务都在容器之中运行。

总之，本章会为你提供一些能替代传统操作系统来运行 Docker 的选择，并提出一些解决方案，在这些方案的主机中，一切都是以一个容器的方式在运行。需要注意的是，除此之外你还有其他选择，比如 VMware Photon（https://vmware.github.io/photon/）。此外，还有一些项目利用了传统的虚拟化技术但是采用了不同的方法，比如 Clear Linux 项目（https://clearlinux.org）和 hyper.sh（https://hyper.sh）。

CoreOS 是一个新的 Linux 发行版，一些公共云服务提供商已经提供了对 CoreOS 的支持。这是一种新的趋势，其目的是建立一个只提供在容器内运行应用程序功能的最简操作系统。理论上讲，它试图通过一个可扩展的、易于管理的操作系统来简化基础设施的运维，这个操作系统能清晰地将应用程序和运维的关注点进行分离。

6.1　在Vagrant中体验CoreOS Linux发行版

6.1.1　问题

你想使用 CoreOS Linux 发行版来运行 Docker 容器，但是在这之前，你想先在本地尝试一下 CoreOS。

6.1.2　解决方案

使用 Vagrant（http://vagrantup.com）在 VirtualBox 中启动一个虚拟机，虚拟机将运行 CoreOS。Vagrant 的官方文档（https://coreos.com/docs/running-coreos/platforms/vagrant/）详细描述了整个操作过程，本范例是官方文档的总结。

第一次通过 Vagrant 启动 CoreOS 虚拟机，需要先克隆一个 Git 仓库（https://github.com/coreos/coreos-vagrant.git），然后运行 `vagrant up` 命令。之后，你将能够通过 `ssh` 连接到已启动的实例并使用 Docker，如下所示。

```
 $ git clone https://github.com/coreos/coreos-vagrant.git
 $ cd coreos-vagrant/
 $ tree
 .
 ├── CONTRIBUTING.md
 ├── MAINTAINERS
 ├── README.md
 ├── Vagrantfile
 ├── config.rb.sample
 └── user-data.sample

0 directories, 6 files
$ vagrant up
$ vagrant ssh
Last login: Mon Jan 12 10:39:30 2015 from 10.0.2.2
CoreOS alpha (557.0.0)
core@core-01 ~ $ docker ps
CONTAINER ID    IMAGE     COMMAND     CREATED     STATUS      PORTS       NAMES
```

CoreOS 使用 `systemd`（http://www.freedesktop.org/wiki/Software/systemd/）作为 Linux 初始

化系统，致力于成为一个最小的发行版，并提供滚动升级功能，可以轻松地实现回滚。核心软件包应直接安装在系统中，应用程序则应完全部署在容器中。因此，在 CoreOS 中并没有包管理器。在 CoreOS 实例中运行的所有服务都以 `systemd` 单元文件运行，并且可以使用诸如 `systemctl` 或 `journalctl` 这样的命令来与这些服务进行交互（https://coreos.com/docs/launching-containers/launching/getting-started-with-systemd），如下所示。

```
$ systemctl list-units | grep docker |awk {'print $1'}
sys-devices-virtual-net-docker0.device
sys-subsystem-net-devices-docker0.device
var-lib-docker-btrfs.mount
docker.service
docker.socket
early-docker.target

$ journalctl -u docker.service
-- Logs begin at Mon 2015-01-12 10:39:15 UTC, ... --
Jan 12 10:39:34 core-01 systemd[1]: Starting Docker ...
Jan 12 10:39:34 core-01 systemd[1]: Started Docker ...
Jan 12 10:39:34 core-01 dockerd[876]: ... msg="+job serveapi(fd://)"
Jan 12 10:39:34 core-01 dockerd[876]: ... msg="+job init_networkdriver()"
Jan 12 10:39:34 core-01 dockerd[876]: ... msg="Listening for HTTP on fd ()"
Jan 12 10:39:34 core-01 dockerd[876]: ... msg="-job init_networkdriver() = OK (0)"
Jan 12 10:39:34 core-01 dockerd[876]: ... msg="Loading containers: start."
Jan 12 10:39:34 core-01 dockerd[876]: ... msg="Loading containers: done."
Jan 12 10:39:34 core-01 dockerd[876]: ... msg="docker daemon: 1.4.1 ..."
Jan 12 10:39:34 core-01 dockerd[876]: ... msg="+job acceptconnections()"
Jan 12 10:39:34 core-01 dockerd[876]: ... msg="-job acceptconnections() = OK (0)"
Jan 12 10:39:34 core-01 dockerd[876]: ... msg="GET /v1.16/containers/json"
Jan 12 10:39:34 core-01 dockerd[876]: ... msg="+job containers()"
Jan 12 10:39:34 core-01 dockerd[876]: ... msg="-job containers() = OK (0)"
```

6.1.3 讨论

尽管可以克隆上面的 Git 仓库，然后通过 `vagrant up` 来启动一个 CoreOS 实例，不过你可能会注意到 config.rb.sample 和 user-data.sample 这两个文件。这两个文件用来配置一个 CoreOS 实例的集群（参见范例 6.3），并在启动时对服务进行初始化设置。Vagrant 会在 Vagrantfile 文件中读取这两个文件，如下所示。

```
CLOUD_CONFIG_PATH = File.join(File.dirname(__FILE__), "user-data")
CONFIG = File.join(File.dirname(__FILE__), "config.rb")
```

举例来说，如果想远程访问在 CoreOS 实例中运行的 Docker 服务，需要将 config.rb.sample 文件复制为 config.rb，将 user-data.sample 复制为 user-data；然后编辑 config.rb 文件，取消对 `$expose_docker_tcp=2375` 这一行的注释，如下所示。

```
$ cp config.rb.sample config.rb
$ cp user-data.sample user-data
$ tree
.
├── CONTRIBUTING.md
├── MAINTAINERS
```

```
├── README.md
├── Vagrantfile
├── config.rb
├── config.rb.sample
├── user-data
└── user-data.sample

0 directories, 8 files
$ vi config.rb #uncomment $expose_docker_tcp=2375
$ vagrant up
```

如果你按照本范例解决方案中的指令创建的 CoreOS 实例还在运行中，那么可以通过 `vagrant reload --provision` 对这个实例进行重新初始化，或者先通过 `vagrant destroy` 销毁这个实例，然后通过 `vagrant up` 来重新创建这个实例。

Vagrant 会为 CoreOS 实例配置一个 NAT 和主机网络接口，并将会在 NAT 网络接口上对 2375 端口进行转发，这样你就可以通过 localhost 连接到 Docker 服务了。

```
$ docker -H tcp://127.0.0.1:2375 ps
CONTAINER ID    IMAGE   COMMAND   CREATED   STATUS    PORTS    NAMES
```

6.1.4 参考

- CoreOS 文档（https://coreos.com/docs/）

讨论 Docker 容器与 CoreOS 专用的 Rocket（https://coreos.com/blog/rocket/）的对比已经超越了本书的范围。Rocket 是 CoreOS 提出的一个 App Container 规范（https://github.com/appc/spec/blob/master/SPEC.md）的实现。在范例 4.6 中，我们将这两种容器格式结合起来，讨论了开放容器组织所做的工作。

6.2 使用`cloud-init`在CoreOS上启动容器

6.2.1 问题

知道了如何通过 Vagrant 启动一个 CoreOS 实例，你想使用 `cloud-init` 工具（https://cloudinit.readthedocs.org/en/latest/）在系统启动时运行一个容器。

6.2.2 解决方案

你知道如何通过 Vagrant 来启动一个 CoreOS 实例（参见范例 6.1）。现在你需要在 user-data 文件中添加一个 `systemd` 单元。CoreOS 将会在系统启动时自动启动这个单元。

创建一个新的 user-data 文件，文件内容如下所示。

```
#cloud-config

coreos:
  units:
    - name: es.service
      command: start
      content: |
        [Unit]
        After=docker.service
        Requires=docker.service
        Description=starts Elastic Search container

        [Service]
        TimeoutStartSec=0
        ExecStartPre=/usr/bin/docker pull dockerfile/elasticsearch
        ExecStart=/usr/bin/docker run -d -p 9200:9200 -p 9300:9300 \
              dockerfile/elasticsearch
```

如果你在范例 6.1 中创建的 CoreOS 实例还在运行中，请先通过 `vagrant destroy` 命令销毁这个实例，然后运行 `vagrant up` 重新创建一个新实例。

> `docker.service` 单元会在 CoreOS 开机时自动启动，所以不需要在 user-data 文件中指定它。

虚拟机会很快启动，然后运行配置文件中定义的 `es.service` 服务。Dcoker 会开始拉取 dockerfile/elasticsearch 镜像。这一操作可能会花费一些时间（所以你需要耐心等待），并会通过 `docker images` 来监控镜像的下载状态。镜像下载完成之后，就会启动一个新的容器（参见 user-data 文件中 `ExecStart` 指令的那一行)，如下所示。

```
$ docker ps
CONTAINER ID     IMAGE                             COMMAND                ...
fa9ff4f2234c     dockerfile/elasticsearch:latest “/elasticsearch/bin/   ...
```

找到虚拟机主机网络接口（即 `eth1`）的 IP 地址，然后打开浏览器或者通过 `curl` 来访问该地址的 9200 端口，如下所示。

```
$ curl -s http://172.17.8.101:9200 | python -m json.tool
{
    "cluster_name": "elasticsearch",
    "name": "Wyatt Wingfoot",
    "status": 200,
    "tagline": "You Know, for Search",
    "version": {
        "build_hash": "89d3241d670db65f994242c8e8383b169779e2d4",
        "build_snapshot": false,
        "build_timestamp": "2014-10-26T15:49:29Z",
        "lucene_version": "4.10.2",
        "number": "1.4.1"
    }
}
```

恭喜，你已经成功在 CoreOS 实例上通过 `cloud-init` 定义了一个 `systemd` 单元，并启动了一个 Elasticsearch（http://www.elasticsearch.com）容器。

6.2.3 讨论

CoreOS 使用了 `coreos-vagrant` 仓库里的 user-data 文件来对实例进行配置，不过使用的是 CoreOS 版本的 `cloud-init`（https://cloudinit.readthedocs.org/en/latest/）。`cloud-init` 已经被很多公有云服务提供商采用，很多 IaaS 软件解决方案也提供了对 `cloud-init` 的支持。`cloud-init` 可以用在云计算服务中，在启动时对启动的虚拟机实例进行上下文环境设置。本范例的一个有趣的部分是，容器是以 `systemd` 单元文件的形式定义的，并随着系统的启动而自动开始运行。CoreOS 官方提供了很多关于这一特性的文档（https://coreos.com/docs/launching-containers/launching/getting-started-with-systemd/）。

CoreOS 拥有自己的 `cloud-init`（https://github.com/coreos/coreos-cloudinit/blob/master/Documentation/cloud-config.md#coreos-parameters）实现，一些 `cloud-init` 操作可能不受支持，其他的则只适用于 CoreOS（比如 `fleet`、`etcd` 和 `flannel`）。

6.3 通过Vagrant启动CoreOS集群，在多台主机上运行容器

6.3.1 问题

你希望更深入地掌握一些 CoreOS 功能和插件（比如 `etcd` 和 `fleet`），以管理一个由多台 Docker 主机组成的集群。

6.3.2 解决方案

如果你还没有进行以下操作，请先从 GitHub 克隆 CoreOS Vagrant 项目，并修改配置文件。

```
$ git clone https://github.com/coreos/coreos-vagrant.git
$ cd coreos-vagrant/
$ cp config.rb.sample config.rb
$ cp user-data.sample user-data
```

我们还可以继续使用范例 6.1 中的 Vagrantfile 文件，然后在 config.rb 文件中设置集群中的实例数量。这个集群由一组 CoreOS 实例组成，这些实例可以由 Vagrant 在 VirtualBox 中启动，也可以在 VMware Fusion 中运行。

在范例 6.2 中，你已经看到了如何修改用户数据来让一个容器在系统启动时就开始运行。在范例 6.1 中，通过修改 config.rb 文件，将 2375 端口暴露出来，以允许远程访问 Docker 守护进程。要想通过 Vagrant 启动一个 CoreOS 集群，你需要修改 config.rb 文件来设置集群中实例的个数。比如，`$num_instances=4` 将会启动四个 CoreOS 实例。

另外，在 config.rb 文件顶部，你会看到一些 Ruby 代码，这些代码会修改 user-data 文件，将 `discovery` 密钥保存到这个 YAML 文件中。这里使用了由 CoreOS 团队运营的发现服务，用来帮助在集群节点上运行 `etcd` 服务。`etcd`（https://github.com/coreos/etcd）是一个高可用的键值存储服务，用于进行配置信息的共享和服务发现，它可以与 CoreOS 结合使用。`etcd` 和其他服务发现解决方案类似，比如 Apache ZooKeeper（http://zookeeper.apache.org）和 Consul（https://consul.io）。你可以让 `etcd` 在不同的主机上运行，但是，在这个范例中，我们将会利用 Vagrantfile 的优点，定义一个在多台主机上运行的 `etcd`，并让它在将要启动的集群之上运行。`etcd` 允许 Docker 主机发现自己，并帮助对容器进行调度。

> 讨论 `etcd` 已经超出了本书的范围。CoreOS 提供了一个简单易用的基于 `etcd` 的发现服务（https://coreos.com/docs/cluster-management/setup/cluster-discovery），以方便启动 CoreOS 集群。Vagrant 的例子中虽然使用了该服务发现方式，但不建议在生产环境中使用。

在 config.rb 文件中，删除该脚本对开始部分代码的注释，并设置集群中实例的个数。

```
if File.exists?('user-data') && ARGV[0].eql?('up')
  require 'open-uri'
  require 'yaml'

  token = open('https://discovery.etcd.io/new').read

  data = YAML.load(IO.readlines('user-data')[1..-1].join)
  data['coreos']['etcd']['discovery'] = token

  yaml = YAML.dump(data)
  File.open('user-data', 'w') { |file| file.write("#cloud-config\n\n#{yaml}") }
end
...
$num_instances=4
```

> 如果你曾经按照范例 6.1 和范例 6.2 进行过操作，那么你需要在启动集群之前，通过 `vagrant destroy` 来销毁所有现存 CoreOS 实例。

在将实例数设置为 4 之后，别忘了将原始的 user-data.sample 文件复制到 user-data，然后只需要执行 `vagrant up` 并等待初始化完成。之后，你就可以 `ssh` 到其中的一个节点，使用一个新工具 `fleet` 列出已经加入到这个集群的所有节点，如下所示。

```
$ cp user-data.sample user-data
$ vagrant up
$ vagrant status
Current machine states:

core-01                   running (virtualbox)
core-02                   running (virtualbox)
```

```
core-03                   running (virtualbox)
core-04                   running (virtualbox)
$ vagrant ssh core-01
CoreOS (stable)
core@core-01 ~ $ fleetctl list-machines
MACHINE   IP    METADATA
01efec94... 172.17.8.102  -
3602cd04... 172.17.8.104  -
cd3de202... 172.17.8.103  -
e4c0e706... 172.17.8.101  -
```

6.3.3 讨论

CoreOS 自己开发的 etcd 服务发现组件在这里用来启动该集群（也就是定义前导符）。在 user-data 文件中，你现在可以看到有一行用来定义 discovery 密钥，这里指定了一个令牌（实际上你的令牌与例子中的令牌会不一样）。

```
discovery: https://discovery.etcd.io/61297b379e5024f33b57bd7e7225d7d7
```

如果通过 curl 命令访问这个 URL（curl -s https://discovery.etcd.io/ 61297b379e5024f33b57bd7e7225d7d7 | python -m json.tool），你将会看到该集群中各节点的 IP 地址。任何人只要得到了你的令牌，就能获得该集群中的节点列表，并能尝试将他的节点添加到你的集群中，所以一定要小心保管自己的令牌。

```
{
    "action": "get",
    "node": {
        "createdIndex": 279743993,
        "dir": true,
        "key": "/_etcd/registry/61297b379e5024f33b57bd7e7225d7d7",
        "modifiedIndex": 279743993,
        "nodes": [
            {
                "createdIndex": 279744808,
                "expiration": "2015-01-19T17:50:15.797821504Z",
                "key": "/_etcd/registry/61297b379e5024f33b57bd7.../e4c0...",
                "modifiedIndex": 279744808,
                "ttl": 599113,
                "value": "http://172.17.8.101:7001"
            },
            {
                "createdIndex": 279745601,
                "expiration": "2015-01-19T17:59:49.196184481Z",
                "key": "/_etcd/registry/61297b379e5024f33b57bd7.../01ef...",
                "modifiedIndex": 279745601,
                "ttl": 599687,
                "value": "http://172.17.8.102:7001"
            },
            {
                "createdIndex": 279746380,
                "expiration": "2015-01-19T17:51:41.963086657Z",
                "key": "/_etcd/registry/61297b379e5024f33b57bd7.../cd3d...",
                "modifiedIndex": 279746380,
```

```
                "ttl": 599199,
                "value": "http://172.17.8.103:7001"
            },
            {
                "createdIndex": 279747319,
                "expiration": "2015-01-19T17:52:33.315082679Z",
                "key": "/_etcd/registry/61297b379e5024f33b57bd7.../3602...",
                "modifiedIndex": 279747319,
                "ttl": 599251,
                "value": "http://172.17.8.104:7001"
            }
        ]
    }
}
```

现在，你的这些节点就组成了一个 `etcd` 集群，这是一个可以完美工作的高可用键值存储服务。通过 `etcdctl` 命令，你可以对键进行 get 和 set 操作，如下所示。

```
core@core-01 ~ $ etcdctl set foobar“Docker”
Docker
core@core-01 ~ $ etcdctl get foobar
Docker
core@core-01 ~ $ etcdctl ls
/foobar
/coreos.com
```

要想在该集群中启动容器，可以像在范例 6.2 中那样定义一个 `systemd` 单元文件，然后通过 `fleetctl` 命令行工具来启动容器（参见范例 6.4）。

6.3.4　参考

- 使用 Vagrant 创建 CoreOS 集群（https://coreos.com/blog/coreos-clustering-with-vagrant/）
- `etcd` 简介（https://coreos.com/docs/distributed-configuration/getting-started-with-etcd/）
- `fleet` 入门（https://coreos.com/docs/launching-containers/launching/launching-containers-fleet/）

6.4　在CoreOS集群上通过fleet启动容器

6.4.1　问题

你有一个可以工作的 CoreOS 集群，想在上面运行容器。

6.4.2　解决方案

当已经有了可工作的 CoreOS 集群时（参见范例 6.3），可以使用 `fleetctl` 命令行工具来启动容器。可以通过编写 `systemd` 单元文件来描述要运行的容器，并通过 `fleetctl start` 命令在集群上对容器进行调度。

比如，回顾一下我们在范例 6.2 中是如何使用 `cloud-init` 来启动容器的。你可以提取出来下面的 `systemd` 单元文件，以在 CoreOS 集群中启动一个 Elasticsearch 容器（我们称之为

es.service），如下所示。

```
[Unit]
After=docker.service
Requires=docker.service
Description=starts Elastic Search container

[Service]
TimeoutStartSec=0
ExecStartPre-=/usr/bin/docker kill es
ExecStartPre-=/usr/bin/docker rm es
ExecStartPre=/usr/bin/docker pull dockerfile/elasticsearch
ExecStart=/usr/bin/docker run --name es -p 9200:9200 \
                                        -p 9300:9300 \
                                        dockerfile/elasticsearch
ExecStop=/usr/bin/docker stop es
```

通过 fleetctl 启动这个容器，如下所示。

```
$ vagrant ssh core-01
$ fleetctl start es.service
$ fleetctl list-units
UNIT    MACHINE       ACTIVE    SUB
es.service  01efec94.../172.17.8.102  activating  start-pre
$ fleetctl list-units
UNIT    MACHINE       ACTIVE  SUB
es.service  01efec94.../172.17.8.102  active  running
```

fleet 将会在集群节点上对这个单元进行调度。systemd 将会开始启动这个 es.service 单元，首先会开始下载相应的镜像。当镜像下载完成后，就会根据单元文件中 ExecStart 的定义启动容器。

6.4.3 讨论

fleet 的命令行工具 fleetctl 也提供了几个方便的命令来检查单元文件的 journal、删除单元文件，以及 ssh 到为该单元分配的节点上。这些命令在调试过程中非常方便，如下所示。

```
$ fleetctl list-units
UNIT    MACHINE       ACTIVE  SUB
es.service  01efec94.../172.17.8.102  active  running
$ fleetctl ssh es.service
Last login: Mon Jan 12 22:03:29 2015 from 172.17.8.101
CoreOS (stable)
core@core-02 ~ $ docker ps
CONTAINER ID    IMAGE                           COMMAND              ...
6fc661ba2153    dockerfile/elasticsearch:latest  "/elasticsearch/bin/ ...
core@core-02 ~ $ exit
$ fleetctl journal es.service
-- Logs begin at Mon 2015-01-12 17:50:47 UTC, end at Mon 2015-01-12 22:13:20 UTC
Jan 12 22:06:13 core-02 ...[node           ] [Wendigo] initializing ...
Jan 12 22:06:13 core-02 ...[plugins        ] [Wendigo] loaded [], sites []
Jan 12 22:06:17 core-02 ...[node           ] [Wendigo] initialized
Jan 12 22:06:17 core-02 ...[node           ] [Wendigo] starting ...
Jan 12 22:06:17 core-02 ...[transport      ] [Wendigo] bound_address ...
```

```
Jan 12 22:06:17 core-02 ... discovery       ] [Wendigo] elasticsearch/_NcgQa...
Jan 12 22:06:21 core-02 ...[cluster.service ] [Wendigo] new_master [Wendigo]...
Jan 12 22:06:21 core-02 ...[http            ] [Wendigo] bound_address ...
Jan 12 22:06:21 core-02 ...[node            ] [Wendigo] started
Jan 12 22:06:21 core-02 ...[gateway         ] [Wendigo] recovered [0] ...
```

6.4.4 参考

- 使用 `fleet`（https://coreos.com/docs/launching-containers/launching/launching-containers-fleet/）启动容器

6.5 在CoreOS实例之间部署flannel覆盖网络

——本范例由 Eugene Yakubovich 提供

6.5.1 问题

你有一个 CoreOS 集群，想使用覆盖网络取代端口转发的方式进行网络通信。

6.5.2 解决方案

在所有 CoreOS 节点上安装 `flannel`。在对 CoreOS 进行初始化的 cloud-config 文件中，加入下面的代码片段。

```
#cloud-config

coreos:
  units:
    - name: flanneld.service
      drop-ins:
        - name: 50-network-config.conf
          content: |
            [Service]
            ExecStartPre=/usr/bin/etcdctl set \
              /coreos.com/network/config \
              '{ "Network": "10.1.0.0/16" }'
      command: start
```

> 请确保所选用的 IP 地址范围在组织内可用。`flannel` 使用 `etcd` 进行协调。请确保已经按照范例的内容建立了一个 `etcd` 集群。

请确保你的安全策略允许基于 UDP 8285 端口的通信。启动 CoreOS 实例，然后等待 `flannel` 完成初始化。

可以使用 `ifconfig` 工具来检查一下 `flannel0` 接口是否已经启动，如下所示。

```
$ ifconfig

flannel0: flags=81<UP,POINTOPOINT,RUNNING>  mtu 1472
        inet 10.1.77.0  netmask 255.255.0.0  destination 10.1.77.0
        unspec 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00  txqueuelen ...
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

接下来，启动一个监听 8000 端口的容器，并让它输出自己的 IP 地址，如下所示。

```
$ docker run -it --rm busybox /bin/sh -c \
    "ifconfig eth0 && nc -l -p 8000"
eth0      Link encap:Ethernet  HWaddr 02:42:0A:01:4D:03
          inet addr:10.1.77.3  Bcast:0.0.0.0  Mask:255.255.255.0
          UP BROADCAST  MTU:1472  Metric:1
          RX packets:3 errors:0 dropped:0 overruns:0 frame:0
          TX packets:1 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:234 (234.0 B)  TX bytes:90 (90.0 B)
```

记下 `ifconfig` 命令返回的 IP 地址，其他位于这个 `flannel` 网络中的容器可以通过这个 IP 地址访问到该容器。

在另一台主机上，启动一个容器并发送一个字符串给上面容器中的监听器，如下所示。

```
$ docker run -it --rm busybox /bin/sh -c \
    "echo Hello, container | nc 10.1.77.3 8000"
```

第一个容器将会输出“Hello, container”，然后退出。

> 如果想要在 cloud-config 文件的 `units` 部分添加更多的内容，需要确保将任何启动 Docker 容器的服务添加到 `flanneld.service` 之后。由于单元文件是按照顺序执行的，这样就能保证在容器启动之前 `flannel` 已经启动完成。

6.5.3 讨论

`flannel` 的配置保存在 `etcd`（/coreos.com/network/config），并且需要在 `flanneld` 启动之前就设置好。最方便的设置方式就是使用 `systemd` 单元定义文件 `flanneld.service` 中的 `ExecStartPre` 指令。像前面所述的那样，这些配置也可以通过 cloud-config 文件写入磁盘。

在真实使用场景下，需要以某种自动方式来分发服务容器的 IP 地址信息。当为服务创建了单元文件之后，可以通过 `etcd` 来注册客户端将要访问的服务器的 IP 地址信息，如下所示。

```
[Service]
ExecStartPre=/usr/bin/docker create --name=netcat-server busybox \
/usr/bin/nc -l -p 8000
ExecStart=/usr/bin/docker start -a netcat-server
ExecStartPost=/bin/bash -c 'etcdctl set /services/netcat-server \
```

```
$(docker inspect --format="{{.NetworkSettings.IPAddress}}" netcat-server)'

ExecStop=/usr/bin/docker stop netcat-server
ExecStopPost=/usr/bin/docker rm netcat-server
```

> 除了使用 `ExecStartPost` 指令，也可以创建一个辅助单元（http://https://coreos.com/fleet/docs/latest/launching-containers-fleet.html）。或者使用 SkyDNS（https://github.com/skynetservices/skydns）项目来为客户端提供 DNS 接口。

默认配置下，`flannel` 会使用一个 TUN 设备来将数据包发送到用户空间来实现 UDP 封装。这是一个成熟的解决方案，因为在很多年前 TUN 设备就已经成为 Linux 内核的一部分了。但是，由于数据包需要经由 `flannel` 守护进程进进出出，所以对性能有很大的影响。现代 Linux 内核已经支持一种称为 VXLAN 的新数据封装技术。VXLAN 也将数据包封装为对网络友好的 UDP，但是这些工作都是在内核中完成的。CoreOS 坚持使用最新的内核，所以可以非常方便地使用 VXLAN。启用 VXLAN 也很简单，只需要在 `flannel` 的配置文件中修改一下 Backend 属性即可，如下所示。

```
ExecStartPre=/usr/bin/etcdctl set /coreos.com/network/config \
    '{ "Network": "10.1.0.0/16", "Backend": { "Type": "vxlan" } }'
```

> 在一个不安全的环境中运行时，最好在 `flannel` 和 `etcd` 通信中采用 TLS 机制。TLS 客户端证书可以用于限制对 `etcd` 的访问。可以参考 `etcd` 和 `flannel` 的文档来获取更详细的信息。

6.6　使用Project Atomic运行Docker容器

6.6.1　问题

你在寻找一个 CoreOS、Ubuntu Snappy 和 RancherOS 之外的操作系统。

6.6.2　解决方案

使用 Project Atomic（http://www.projectatomic.io）。Atomic 是由 Red Hat 资助的项目，并受到了 RHEL 和 CentOS 发行版的启发。Atomic 基于 CentOS 7，与 CoreOS、Ubuntu Snappy 和 RancherOS 一样，它的目标也是提供针对 Docker 进行过优化的 Linux 发行版，以容器的方式来部署应用程序。Atomic 通过称为 `rpm-ostree`（http://www.projectatomic.io/docs/os-updates/）的组件来进行升级。当有新的升级可用时，它就会重启并安装新的升级程序，同时它也支持对升级进行回滚。

可以使用 CentOS 的构建（http://buildlogs.centos.org/rolling/7/isos/x86_64/）来运行 Atomic。可以选择下载 ISO，一个为基于内核的虚拟机（kernel-based virtual machine，KVM）准备

的 qcow2 镜像，或者一个 Vagrant 虚拟机。（为 VirtualBox 准备的 Vagrant 虚拟机可能并不是最新版本的 Atomic，需要进行升级。）

像本书的其他部分一样，为了方便使用，我也准备了一个 Vagrantfile 文件，如下所示。

```
$bootstrap=<<SCRIPT
gpasswd -a vagrant root
SCRIPT

# Vagrantfile API和语法的版本号,不要修改这个版本号,除非你知道怎么去做
VAGRANTFILE_API_VERSION = "2"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
  # 每一个Vagrant虚拟环境都需要使用一个box来构建
  config.vm.box = "atomic"
  config.vm.box_url = "http://buildlogs.centos.org/rolling/7/isos/x86_64/\
                     CentOS-7-x86_64-AtomicHost-Vagrant-VirtualBox.box"

  config.vm.provider "virtualbox" do |vb, override|
     vb.customize ["modifyvm", :id, "--memory", "2048"]
  end

  config.vm.network :forwarded_port, host: 9090, guest: 9090
  config.vm.provision :shell, inline: $bootstrap

end
```

如果已经安装了 Vagrant ，则只需要执行 `vagrant up` 就可以 `ssh` 到这台 Atomic 主机。克隆本书附带的代码仓库，你就能看到上面的 Vagrantfile 文件，如下所示。

```
$ git clone https://github.com/how2dock/docbook
$ cd dockbook/ch06/atomic
$ vagrant up
$ vagrant ssh
```

Atomic 计算机启动之后，里面的 Docker 就已经安装好了。你可以通过 `atomic` 命令来查看 Atomic 主机，使用 `sudo atomic host upgrade` 命令来进行升级。

6.6.3 参考

- Atomic 项目文档（http://www.projectatomic.io/docs/）

6.7 在AWS上启动Atomic实例运行Docker

6.7.1 问题

你不希望用 Vagrant 来尝试 Atomic（参见范例 6.6），同时也不想使用 ISO 镜像。

6.7.2 解决方案

在 Amazon EC2 上运行 Atomic 实例。AWS EC2 提供了 Atomic 的 AMI。可以打开你的 AWS 管理控制台，进入实例启动向导，然后查找名为 atomic 的社区 AMI，你会发现有几个 AMI 结果可用，这些 AMI 大多数都基于 Fedora 22 版本。在创建了 SSH 密钥对之后，就可以启动实例了。当实例启动后，就可以连接到该实例。作为例子，这里假设该实例的 IP 地址为 52.18.234.151。现在你可以访问 Docker 了，如下所示。

```
$ ssh -i ~/.ssh/<SSH_PRIVATE_KEY> fedora@52.18.234.151
[fedora@ip-172-31-46-186 ~]$ sudo docker ps
CONTAINER ID    IMAGE       COMMAND     CREATED     STATUS      PORTS     NAMES
[fedora@ip-172-31-46-186 ~]$ sudo docker version | version
Client version: 1.5.0-dev
...
Server version: 1.5.0-dev
```

这个实例自带了一个 `atomic` 命令，你可以通过该命令对主机进行升级。检查实例的状态并启动升级。此操作将会下载最新版的 Atomic。同时，你需要重启实例才能完成升级。

```
[fedora@ip-172-31-46-186 ~]$ atomic host status
  TIMESTAMP (UTC)      VERSION   ID          OSNAME        REFSPEC
* 2015-05-12 18:53:06  22.66     cd414cba85  fedora-atomic fedora-atomic:...
[fedora@ip-172-31-46-186 ~]$ atomic host upgrade
Updating from: fedora-atomic:fedora-atomic/f22/x86_64/docker-host
[fedora@ip-172-31-46-186 ~]$ sudo systemctl reboot
```

在重启之后，你会发现该主机已经自动升级到了新版本的 Docker，这也是最新版 Atomic 里附带的，如下所示。

```
[fedora@ip-172-31-46-186 ~]$ sudo docker version
Client version: 1.7.1.fc22
...
Server version: 1.7.1.fc22
...
```

6.7.3 讨论

你可以通过 AWS 命令行工具或者本范例提供的脚本来启动实例。本范例附带的脚本有一个好处就是，它使用了 Apache Libcloud，因此很方便移植到其他提供了 Atomic 模板的云计算服务提供商，如下所示。

```
#!/usr/bin/env python

import os
from libcloud.compute.types import Provider
from libcloud.compute.providers import get_driver

ACCESS_ID = os.getenv('AWSAccessKeyId')
SECRET_KEY = os.getenv('AWSSecretKey')

IMAGE_ID = 'ami-dd3fb0aa'
SIZE_ID = 'm3.medium'
```

```
cls = get_driver(Provider.EC2_EU_WEST)
driver = cls(ACCESS_ID, SECRET_KEY)

sizes = driver.list_sizes()
images = driver.list_images()
size = [s for s in sizes if s.id == SIZE_ID][0]
image = [i for i in images if i.id == IMAGE_ID][0]

# 读取云配置文件
userdata = "\n".join(open('./cloud.cfg').readlines())

# 使用你的ssh密钥对名称进行替换
# 你需要在默认的安全组中打开SSH的22端口
# 这里还假定密钥对名称为'atomic'
name = "atomic"
node = driver.create_node(name=name, image=image,size=size,ex_keyname='atomic', \
                          ex_userdata=userdata)
snap, ip = driver.wait_until_running(nodes=[node])[0]
print ip[0]
```

正如这段脚本中注释部分所表明的那样，你需要创建一个开放 22 端口的安全组，一个名为 `atomic` 的 SSH 密钥对，以及一个包括你的用户数据的名为 cloud.cfg 的文件。

6.8 快速体验在Ubuntu Core Snappy上运行Docker

6.8.1 问题

你希望测试一下新发布的 Ubuntu Core Snappy。你不想连接到公有云，也不想手动安装 ISO 文件，还希望避免阅读大量的文档。你只是想快速尝试一下 Snappy。

6.8.2 解决方案

我准备了一个 Vagrantfile 文件，可以使用这个文件在你的宿主机上启动一个 Ubuntu Core Snappy 虚拟机。如果还没有克隆本书附带的代码仓库，就需要先克隆一下该仓库。然后，进入 ch06/snappy 文件夹，执行 `vagrant up` 命令。最后，`ssh` 到这个 VM 并使用 Docker，如下所示。

```
$ git clone https://github.com/how2dock/docbook.git
$ cd docbook/ch06/snappy
$ vagrant up
$ vagrant ssh
$ snappy info
release: ubuntu-core/devel
frameworks: docker
apps:
$ docker ps
CONTAINER ID    IMAGE   COMMAND     CREATED     STATUS      PORTS       NAMES
```

这一过程会从 Atlas 上下载 komljen/ubuntu-snappy（https://vagrantcloud.com/komljen/boxes/ubuntu-snappy）镜像。如果你不信任这个镜像，可以不使用。

目前的 Ubuntu Snappy 为 alpha 版，可以将其视为技术预览版。

6.8.3　讨论

2014 年 12 月 9 日，Canonical 发布了 Snappy（https://insights.ubuntu.com/2014/12/09/a-new-transactionally-updated-snappy-ubuntu-core/），一个基于 Ubuntu Core 的、支持事务更新的新 Linux 发行版。这与迄今为止的主流 Ubuntu 服务器或者桌面系统中的软件包或者应用管理模型有显著的不同。

Ubuntu Core（https://wiki.ubuntu.com/Core）是一个精简的 root 文件系统，为安装软件包提供了完整的操作系统功能。使用 Snappy，你可以在 Ubuntu Core 中进行事务更新，或者进行回滚。Ubuntu Core 的实现，参考了 Ubuntu 手机应用管理系统中基于镜像的工作流程。也就是说，`apt-get` 不能在 `snappy` 中使用。

这也使得 Docker 成为 Snappy 上最合适的应用框架。Docker 作为框架安装，并能够以原子方式进行升级或者回滚。

按照下面这个示例进行操作。

```
$ apt-get update
Ubuntu Core does not use apt-get, see 'snappy --help'!
$ snappy --help
...
Commands:
  {info,versions,search,update-versions,update,
   rollback,install,uninstall,tags,build,chroot,
   framework,fake-version,nap}
    info
    versions
    search
    update-versions
    update
    rollback            undo last system-image update.
    install
    uninstall
    tags
    build
    chroot
    framework
...
```

```
$ snappy versions
Part         Tag   Installed  Available  Fingerprint     Active
ubuntu-core  edge  140        142        184ad1e863e947  *
```

为了运行 Docker，需要安装称为 snappy 的框架。可以像下面这样查找并安装 Docker 框架。

```
$ snappy search docker
Part    Version    Description
docker  1.3.2.007  The docker app deployment mechanism
$ sudo snappy install docker
docker      4 MB     [===============]    OK
Part    Tag   Installed  Available  Fingerprint     Active
docker  edge  1.3.2.007  -          b1f2f85e77adab  *
$ snappy versions
Part         Tag   Installed  Available  Fingerprint     Active
ubuntu-core  edge  140        142        184ad1e863e947  *
docker       edge  1.3.2.007  -          b1f2f85e77adab  *
```

现在就可以在 Ubuntu Snappy 上使用 Docker 了，如下所示。

```
$ docker ps
CONTAINER ID    IMAGE    COMMAND     CREATED     STATUS       PORTS       NAMES
```

在 Ubuntu Snappy 上享受运行 Docker 的乐趣吧。

6.8.4 参考

- Snappy 公告（https://insights.ubuntu.com/2014/12/09/a-new-transactionally-updated-snappy-ubuntu-core/）
- 命令行示例（http://blog.dustinkirkland.com/2014/12/its-a-snap.html）

6.9 在AWS EC2上启动Ubuntu Core Snappy实例

6.9.1 问题

你已经通过 Vagrant 体验了一下 Ubuntu Snappy（参见范例 6.8），但是你想在公有云，尤其是 AWS EC2 上运行 Snappy。

6.9.2 解决方案

本范例涉及一些高级内容，这需要你具备一定的关于 Amazon AWS 的知识。尽管这里列出了所有的操作步骤，但是在开始本范例之前，你可能还需要阅读一下 James Murty 编写的 *Programming Amazon Web Services*（http://shop.oreilly.com/product/9780596515812.do）。

作为先决条件，你需要满足以下要求。

- 一个 AWS 账号（http://aws.amazon.com）
- 一组访问和安全 API 密钥（http://docs.aws.amazon.com/general/latest/gr/getting-aws-sec-creds.html）
- 一个默认的允许 SSH 连接的 AWS 安全组
- 一组名为 `snappy` 的密钥对
- 一台安装了 Apache Libcloud（http://libcloud.apache.org）的主机（`sudo pip install apache-libcloud`）

为了尽可能方便操作，我准备了一个 Python 脚本，这个脚本使用 Apache Libcloud 来启动一台 Amazon EC2 实例。Libcloud 是一个 API 包装器，它抽象了各云计算提供商 API 之间的差异。你也可以通过简单修改这个脚本来在其他云计算提供商的平台上启动 Snappy 实例。如果你已经满足了前面所提到的要求，那么就可以进行如下操作了。

```
$ git clone https://github.com/how2dock/docbook
$ cd ch06/snappy-cloud
$ ./ec2snappy.py
54.154.68.31
```

当云主机实例启动后，可以 `ssh` 到实例中并检查 Snappy 的版本，如下所示。

```
$ ssh -i ~/.ssh/id_rsa_snappy ubuntu@54.154.68.31
$ snappy versions
Part         Tag   Installed  Available  Fingerprint     Active
ubuntu-core  edge  141        142        7f068cb4fa876c  *
```

剩下的工作就是安装 Docker Snappy 框架，然后就可以启动容器了，如下所示。

```
$ snappy search docker
Part    Version    Description
docker  1.3.2.007  The docker app deployment mechanism
$ sudo snappy install docker
docker      4 MB     [===============]    OK
Part    Tag   Installed  Available  Fingerprint     Active
docker  edge  1.3.2.007  -          b1f2f85e77adab  *
$ docker pull ubuntu:14.04
ubuntu:14.04: The image you are pulling has been verified
511136ea3c5a: Pull complete
3b363fd9d7da: Pull complete
607c5d1cca71: Pull complete
f62feddc05dc: Pull complete
8eaa4ff06b53: Pull complete
Status: Downloaded newer image for ubuntu:14.04
$ docker images
REPOSITORY      TAG         IMAGE ID        CREATED         VIRTUAL SIZE
ubuntu          14.04       8eaa4ff06b53    9 days ago      192.7 MB
```

上面的简单 Python 脚本中用到了 Libcloud。这假设你已经将你的 AWS 密钥设置为 `AWSAccessKeyId` 和 `AWSSecretKey` 中的环境变量。这个脚本会在 `eu_west_1` 可用区内创建一个 `m3.medium` 类型的实例，并且允许通过 SSH 连接到该实例。最后，这个脚本将 SSH 密

钥对设置为 snappy（你需要在运行该脚本之前创建这个密钥对，并将私钥保存为 ~/.ssh/id_rsa_snappy），如下所示。

```
#!/usr/bin/env python

import os
from libcloud.compute.types import Provider
from libcloud.compute.providers import get_driver

ACCESS_ID = os.getenv('AWSAccessKeyId')
SECRET_KEY = os.getenv('AWSSecretKey')

IMAGE_ID = 'ami-20f34b57'
SIZE_ID = 'm3.medium'

cls = get_driver(Provider.EC2_EU_WEST)
driver = cls(ACCESS_ID, SECRET_KEY)

sizes = driver.list_sizes()
images = driver.list_images()

size = [s for s in sizes if s.id == SIZE_ID][0]
image = [i for i in images if i.id == IMAGE_ID][0]

#读取云配置文件
userdata = "\n".join(open('./cloud.cfg').readlines())

#使用你的ssh密钥对名称进行替换
#你需要在默认的安全组中打开SSH的22端口
name = "snappy"
node = driver.create_node(name=name, image=image,size=size, \
                          ex_keyname='snappy',ex_userdata=userdata)
print node.extra['network_interfaces']
```

如果你想使用 EU_WEST 之外的可用区，需要在 Snappy 的公告（http://www.ubuntu.com/cloud/tools/snappy）中确认你想要使用的可用区及其对应的 AMI ID。

6.9.3 讨论

Snappy 现在提供了对 Amazon AWS、Google GCE 和 Microsoft Azure beta 版的支持，如图 6-1 所示。

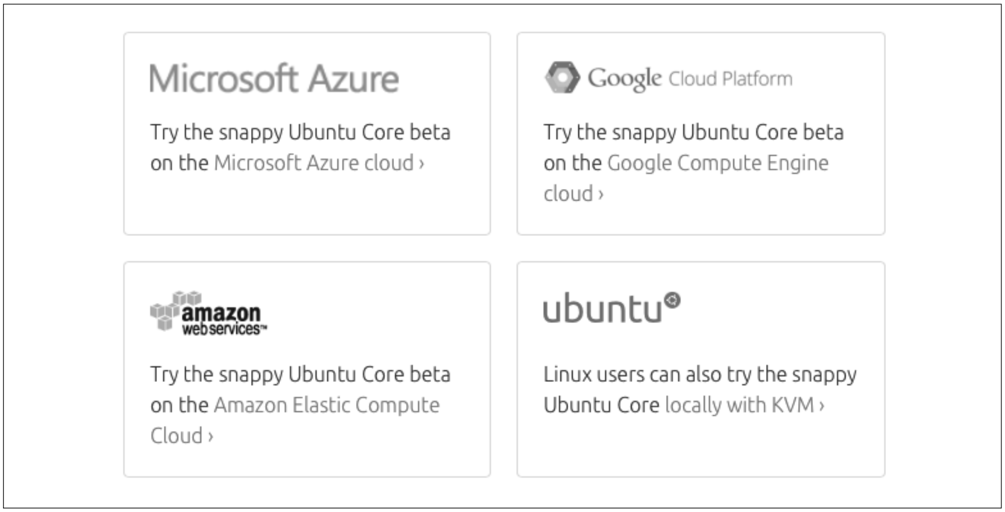


图 6-1：Snappy 公有云 beta 版

你可以按照 Snappy 说明文档（http://www.ubuntu.com/cloud/tools/snappy）在这些公有云中使用各提供商的命令行工具创建云主机实例，或者修改上面提供的基于 Libcloud 的脚本。

比如在 Google GCE（https://cloud.google.com/compute/）上，创建了账号并安装了 Cloud SDK（https://cloud.google.com/sdk/）之后，你就可以通过 GCE Cloud SDK 创建 Snappy 实例了，如下所示。

```
$ gcloud compute instances create snappy-test \
         --image-project ubuntu-snappy \
         --image ubuntu-core-devel-v20141215 \
         --metadata-from-file user-data=cloud.cfg
Created [https://www.googleapis.com/compute/v1/projects/runseb/zones/\
         europe-west1-c/instances/snappy-test2].
NAME         ZONE           MACHINE_TYPE  INTERNAL_IP   EXTERNAL_IP    STATUS
snappy-test2 europe-west1-c n1-standard-1 10.240.250.42 130.211.103.14 RUNNING
$ ssh -i ~/.ssh/id_rsa_snappy ubuntu@130.211.103.14
...
$ snappy info
release: ubuntu-core/devel
frameworks:
apps:
```

享受云中的 Snappy 吧！

6.9.4　参考

- 使用 EC2 工具的详细步骤（http://www.ubuntu.com/cloud/tools/snappy）
- Snappy 支持 AWS 的公告（http://blog.dustinkirkland.com/2014/12/awsnap-snappy-ubuntu-now-available-on.html）

6.10　在RancherOS中运行Docker容器

6.10.1　问题

你在寻找一个 CoreOS、Ubuntu Snappy 和 Project Atomic 之外的操作系统。

6.10.2　解决方案

尝试 Rancher Labs（http://rancher.com）最新发布的 RancherOS（http://rancher.com/rancher-os/）。RancherOS 是一个精简版 Linux 发行系统，只有 20 MB 大小。RancherOS 的所有组件都是一个 Linux 容器，它移除了 `systemd` 初始化系统，而是运行一个称为 `system-docker` 的守护进程，将其作为 PID 为 1 的进程，并且直接在容器中运行 Linux 服务。`system-docker` 接着会启动 Docker 守护进程，用于运行应用程序容器。

> RancherOS 发布不久（http://rancher.com/announcing-rancher-os/），还处于开发阶段。

为了便于测试，RancherOS 提供了一个方便的 Vagrant 项目（https://github.com/rancherio/os-vagrant）。下面的四行 bash 脚本将会帮助你启动并运行 RancherOS。

```
$ git clone https://github.com/rancherio/os-vagrant
$ cd os-vagrant
$ vagrant up
$ vagrant ssh
```

然后你就可以在这台计算机上使用最新版的 Docker 了，如下所示。

```
[rancher@rancher ~]$ docker ps
CONTAINER ID    IMAGE    COMMAND    CREATED    STATUS    PORTS    NAMES
[rancher@rancher ~]$ docker version
Client version: 1.7.0
...
Server version: 1.7.0
...
```

作为 root 用户，你可以使用 `system-docker` 命令查看在容器中运行的系统服务，如下所示。

```
[rancher@rancher ~]$ sudo system-docker ps
CONTAINER ID     IMAGE                       COMMAND               ... NAMES
bde437da2059     rancher/os-console:v0.3.3   "/usr/sbin/entry.sh       console
2113b2e191ea     rancher/os-ntp:v0.3.3       "/usr/sbin/entry.sh       ntp
a7795940ec89     rancher/os-docker:v0.3.3    "/usr/sbin/entry.sh       docker
b0266396e938     rancher/os-acpid:v0.3.3     "/usr/sbin/entry.sh       acpid
aa8e18e59e67     rancher/os-udev:v0.3.3      "/usr/sbin/entry.sh       udev
f7145dfd21c9     rancher/os-syslog:v0.3.3    "/usr/sbin/entry.sh       syslog
```

6.10.3　讨论

你也可以在 Amazon EC2 中找到 RancherOS 的 AMI（https://github.com/rancherio/os）。

6.10.4　参考

- RancherOS 的 GitHub 主页（https://github.com/rancherio/os）

第 7 章

Docker生态环境：工具

7.0 简介

Docker 本身非常强大，我们已经学习了关于 Docker 的所有主题，这些内容会为你带来一种全新的编写分布式应用程序的方式。但是，其庞大而活跃的生态系统让 Docker 显得更加强大。本章会对 Docker 生态系统的众多工具进行讲解。

一开始我们会介绍 Docker 生态系统中那些来自 Docker 公司本身的工具。如果你已经安装了第 1 章中介绍的 Docker Toolbox，那么这些工具应该已经安装到了你的系统之中。如果还没有安装 Docker Toolbox，本章相关的范例也将分别介绍如何安装这些工具。首先，在范例 7.1 中，我们会介绍如何使用 `docker-compose`。Docker Compose 通过一个 YAML 文件来描述一个多容器应用程序。我们会介绍本章的第一个例子 WordPress，并讲解 Docker Compose 的配置文件，这个配置文件通过使用两个容器来运行一个 WordPress 站点。之后，在范例 7.2 中，我们会通过一个更复杂的 Compose 例子，来看一下如何部署一个单节点 Mesos（http://mesos.apache.org）集群。在 Compose 之后，我们会在范例 7.3 中介绍 Swarm。作为 Docker 的一个集群管理软件，Swarm 允许你在单一 Docker API 接口下，暴露多台 Docker 主机。从客户端来说，所有的一切看起来与单台 Docker 主机的设置相同，但是 Swarm 可以管理多台 Docker 主机，并在这些 Docker 主机中对容器进行调度。在范例 7.4 中，我们会讲解如何使用 `docker-machine` 来轻松创建你的 Docker Swarm 集群，第 1 章已经对 `docker-machine` 作了介绍。这个工具非常方便，你可以使用 `docker-machine` 在公有云中创建多台 Docker 主机，并自动将它们配置为一个 Swarm 集群。作为对 Docker 公司提供的工具的总结，最后我们会在范例 7.5 中简要介绍一下 Kitematic，这是 Docker 的一个桌面用户界面。

除了 Docker 公司之外，还有大量的项目一起构成了 Docker 的生态系统，本章的余下部

分将会对其中一些项目进行介绍。首先，在范例 7.6 中会介绍一个基本 Docker Web 界面，然后在范例 7.7 中会介绍一个基于 `docker-py` 的交互式 shell。如果你是熟悉配置管理工具的系统管理员，那么对范例 7.8 应该会比较感兴趣，这一范例将会介绍如何使用 Ansible playbook 管理容器的部署。

对容器进行编排也是 Docker 中比较有意思的一个话题。本章会涉及的 Docker Swarm 就是一个容器编排工具，实际上，除此之外还有很多其他的编排工具。在范例 7.9 中我们会介绍 Rancher，这个编排工具有很多有趣的功能，比如多数据中心网络、负载平衡以及与 Docker Compose 的集成等。范例 7.10 中会介绍 CloudFoundry Lattice，这是一个学习 CloudFoundry 如何管理微服务应用的非常好的入门材料，并且它也是 Docker 兼容的。我们还会深入学习 Mesos，首先在范例 7.11 中会介绍一下如何构建一个单节点 Mesos 沙箱，然后在范例 7.12 再来介绍如何构建一个 Mesos 集群。

在本章最后，范例 7.13 会介绍基于 `registrator` 的服务自发现机制。当有大量的临时容器在集群上运行时，你会希望有一个系统可以用来监测容器，并将它们注册到数据存储服务中。这个数据存储服务还可以用于服务定位，确保你的应用程序保持运行状态。前面提到的一些编排工具都提供了服务发现机制，但是如果你想构建自己的编排工具，`registrator` 将会是一个不错的解决方案。

7.1 使用Docker Compose创建WordPress站点

7.1.1 问题

你已经按照范例 1.16 创建了一个 WordPress 站点，但是你想通过一种更清晰的描述文件来描述这个多容器的配置，并通过一条命令就能启动这些容器。

7.1.2 解决方案

使用 Docker Compose（https://docs.docker.com/compose/），它是一个用于定义和运行多容器 Docker 应用程序的命令行工具。使用 Docker Compose，你可以在 YAML 文件中定义要运行的服务，然后通过 `docker-compose` 命令来启动服务。

首先，你需要安装（https://docs.docker.com/compose/）Docker Compose。你可以通过 Python 索引服务或者一条 `curl` 命令来安装 Docker Compose。

如果使用的是自己的 Docker 主机，可以使用 `pip` 从 Python 索引服务手动安装 Docker Compose，如下所示。

```
$ sudo apt-get install python-pip
$ sudo pip install -U docker-compose
```

或者通过 `curl` 命令，如下所示。

```
$ curl -L https://github.com/docker/compose/releases/download/1.4.0/\
        docker-compose-`uname -s`-`uname -m` > /usr/local/bin/docker-compose
```

如果是使用我提供的例子，你只需要在使用 Docker Compose 之前执行 `vagrant up` 即可，如下所示。

```
$ git clone https://github.com/how2dock/docbook.git
$ cd docbook/ch07/compose/
$ vagrant up
$ vagrant ssh
$ docker-compose --version
docker-compose 1.4.0
```

接下来要做的是在 YAML 文件中定义组成 WordPress 应用的两个容器服务。每个服务都在一个容器中运行。你需要为每个服务命名。在这个例子中，你将 WordPress 服务命名为 `wordpress`，将 MySQL 服务命名为 `db`。每个服务还需要包括镜像的定义。范例 1.16 中使用的命令行参数也都需要在 YAML 文件中定义：暴露的端口号、环境变量和已挂载的卷。

创建如下的 docker-compose.yml 文本文件。（如果你使用了 Vagrant 虚拟机，这个文件已经被保存到了 /vagrant/docker-compose.yml，找到该文件。）

```
wordpress:
  image: wordpress
  links:
   - mysql
  ports:
   - "80:80"
  environment:
   - WORDPRESS_DB_NAME=wordpress
   - WORDPRESS_DB_USER=wordpress
   - WORDPRESS_DB_PASSWORD=wordpresspwd
mysql:
  image: mysql
  volumes:
   - /home/docker/mysql:/var/lib/mysql
  environment:
   - MYSQL_ROOT_PASSWORD=wordpressdocker
   - MYSQL_DATABASE=wordpress
   - MYSQL_USER=wordpress
   - MYSQL_PASSWORD=wordpresspwd
```

要想启动这两个容器，你需要到 docker-compose.yml 文件所在目录下，在命令行中键入 `docker-compose up -d` 命令。这两个容器就会启动，并会通过 Docker 链接而链接起来，之后你就可以通过在浏览器中打开 http://<ip_of_host> 来访问这个 WordPress 站点了，如下所示。

```
$ docker-compose up -d
Creating vagrant_mysql_1...
Creating vagrant_wordpress_1...
$ docker-compose ps
       Name                  Command               State        Ports
----------------------------------------------------------------------------------
vagrant_mysql_1       /entrypoint.sh mysqld         Up      3306/tcp
vagrant_wordpress_1   /entrypoint.sh apache2-for ...   Up      0.0.0.0:80->80/tcp
```

7.1.3 讨论

Docker Compose 的前身为 Orchard 团队开发过 Fig（http://www.fig.sh）。在 Orchard 被 Docker 公司收购后，Fig 也更名为 Docker Compose。尽管现在 Docker Compose 还是独立于 Docker 的可执行文件，但是可以期待将来会与 Docker CLI 进行深度整合。Docker Compose 的源代码也可以在 GitHub（https://github.com/docker/compose）上找到。

Docker Compose 提供了下面一些命令来管理容器环境。

```
Fast, isolated development environments using Docker.
...

Commands:
  build     Build or rebuild services
  help      Get help on a command
  kill      Kill containers
  logs      View output from containers
  port      Print the public port for a port binding
  ps        List containers
  pull      Pulls service images
  rm        Remove stopped containers
  run       Run a one-off command
  scale     Set number of containers for a service
  start     Start services
  stop      Stop services
  restart   Restart services
  up        Create and start containers
```

对于每种命令，都可以通过在其后面加上 `--help` 来获得其帮助信息，比如 `docker-compose kill --help` 会显示 `kill` 命令的使用方法。很多命令都需要指定一个 `SERVICE` 参数，一个服务就是 docker-compose.yml 文件中为要运行的容器定义的名称。比如，你可以停止 WordPress 服务，然后再启动该服务，如下所示。

```
$ docker-compose stop wordpress
Stopping vagrant_wordpress_1...
$ docker-compose ps
       Name                      Command               State     Ports
-----------------------------------------------------------------------
vagrant_mysql_1       /entrypoint.sh mysqld            Up       3306/tcp
vagrant_wordpress_1   /entrypoint.sh apache2-for ...   Exit 0
$ docker-compose start wordpress
Starting vagrant_wordpress_1...
$ docker-compose ps
       Name                      Command               State         Ports
--------------------------------------------------------------------------------
vagrant_mysql_1       /entrypoint.sh mysqld            Up      3306/tcp
vagrant_wordpress_1   /entrypoint.sh apache2-for ...   Up      0.0.0.0:80->80/tcp
```

7.2 使用Docker Compose在Docker上对Mesos和Marathon进行测试

7.2.1 问题

你对 Apache Mesos（http://mesos.apache.org）很感兴趣。这是一个数据中心资源分配系统，Twitter 等公司都在使用 Mesos。为了实现服务器利用率最大化，Mesos 可以在不同类型的工作负载之间进行多层次的调度来共享资源。在将 Mesos 投入到生成环境之前，你希望先在一台服务器上试用一下 Mesos。

7.2.2 解决方案

使用 Docker Compose（参见范例 7.1）在一台 Docker 主机上通过一条命令部署 Mesos，非常简单。

你需要启动四个容器：一个用于运行 ZooKeeper（http://zookeeper.apache.org），一个用于运行 Mesos 主节点，一个用于 Mesos 从属节点，还有一个用于运行 Mesos Marathon（https://github.com/mesosphere/marathon）框架。在 Docker Compose 的 YAML 配置文件中定义这四个服务以及它们的启动参数。可以方便地启动这四个容器。下面是一个通过 Docker Compose 部署 Mesos 的 YAML 描述文件的例子。

```
zookeeper:
  image: garland/zookeeper
  ports:
   - "2181:2181"
   - "2888:2888"
   - "3888:3888"
mesosmaster:
  image: garland/mesosphere-docker-mesos-master
  ports:
   - "5050:5050"
  links:
   - zookeeper:zk
  environment:
   - MESOS_ZK=zk://zk:2181/mesos
   - MESOS_LOG_DIR=/var/log/mesos
   - MESOS_QUORUM=1
   - MESOS_REGISTRY=in_memory
   - MESOS_WORK_DIR=/var/lib/mesos
marathon:
  image: garland/mesosphere-docker-marathon
  links:
   - zookeeper:zk
   - mesosmaster:master
  command: --master zk://zk:2181/mesos --zk zk://zk:2181/marathon
  ports:
   - "8080:8080"
mesosslave:
```

```
    image: garland/mesosphere-docker-mesos-master:latest
    ports:
     - "5051:5051"
    links:
     - zookeeper:zk
     - mesosmaster:master
    entrypoint: mesos-slave
    environment:
     - MESOS_HOSTNAME=192.168.33.10
     - MESOS_MASTER=zk://zk:2181/mesos
     - MESOS_LOG_DIR=/var/log/mesos
     - MESOS_LOGGING_LEVEL=INFO
```

> 为了访问 Marathon 沙箱，我们在启动 Mesos 从属节点时指定了环境变量 `MESOS_HOSTNAME=192.168.33.10`。你需要将这个 IP 地址替换为你的 Docker 主机的真实 IP 地址。

将这个文件复制到 docker-compose.yml，然后启动 Docker Compose，如下所示。

```
$ ./docker-compose up -d
Recreating vagrant_zookeeper_1...
Recreating vagrant_mesosmaster_1...
Recreating vagrant_marathon_1...
Recreating vagrant_mesosslave_1...
...
```

当镜像下载完毕，容器启动之后，你就可以通过 http://<IP_OF_HOST>:5050 来访问 Mesos UI 了。Marathon UI 则可以通过该主机的 8080 端口来访问。

7.2.3　讨论

如果你已经克隆了本书附带的在线仓库，只需要执行 `vagrant up` 就可以开始通过 Docker 运行 Mesos 了，如下所示。

```
$ git clone https://github.com/how2dock/docbook.git
$ cd dockbook/ch07/compose
$ vagrant up
$ vagrant ssh
$ cd /vagrant
$ docker-compose -f mesos.yml up -d
```

然后你就可以通过 `docker-compose` 命令来管理容器了。

7.2.4　参考

- 七条命令部署 Mesos（https://medium.com/@gargar454/deploy-a-mesos-cluster-with-7-commands-using-docker-57951e020586#.t0kqmb541）
- Mesos 框架（http://mesos.apache.org/documentation/latest/mesos-frameworks/）

7.3 在Docker Swarm集群上运行容器

7.3.1 问题

你知道如何在单台主机上使用 Docker。你希望能够在由多台主机构成的集群上运行容器，同时保持与在单台主机上使用 Docker CLI 相同的用户体验。

7.3.2 解决方案

使用 Docker Swarm（https://github.com/docker/swarm）。Docker Swarm 是 Docker 原生的集群工具，可以像使用单台 Docker 主机那样访问一组 Docker 主机。Docker Swarm 是在 2014 年 12 月的欧洲 Docker 大会（http://blog.docker.com/tag/docker-swarm/）上发布的。Swarm 最初的 beta 版则是在 2015 年 2 月 26 日发布的（http://blog.docker.com/2015/02/scaling-docker-with-swarm/）。在本书编写之际，Docker Swarm 还处于 beta 版阶段。

为了方便测试 Docker Swarm，我准备了一套 Vagrant 环境和一个初始化脚本，这个脚本会配置一个由四个节点构成的 Swarm 集群。这个集群有一个头节点和三个计算节点，所有节点运行的都是 Ubuntu 14.04 系统。为了启动这个集群，你需要克隆本书附带的 Git 仓库（如果你之前还没有克隆过），然后进入 ch07 文件夹下面的 swarm 子文件夹。之后，使用 Vagrant 这个集群，如下所示。

```
$ git clone https://github.com/how2dock/docbook.git
$ cd docbook/ch07/swarm/
$ vagrant up
```

你会看到 Vagrant 启动了四台虚拟机。这些虚拟机将会使用 Vagrantfile 中定义的 bash 脚本进行初始化，如下所示。

```
...
$bootstrap=<<SCRIPT
apt-get update
curl -sSL https://get.docker.com/ | sudo sh
gpasswd -a vagrant docker
echo "DOCKER_OPTS=\"-H tcp://0.0.0.0:2375\"" >> /etc/default/docker
service docker restart
SCRIPT

$swarm=<<SCRIPT
apt-get update
curl -sSL https://get.docker.com/ | sudo sh
gpasswd -a vagrant docker
docker pull swarm
SCRIPT
...
```

当所有的节点都启动，Vagrant 命令返回之后，你可以通过 `ssh` 连接到头节点，使用 swarm 镜像启动一个 Swarm 容器，这个镜像已经在系统初始化的过程中拉取到本地了。

```
$ vagrant ssh swarm-head
$ docker run -v /vagrant:/tmp/vagrant -p 1234:1234 -d swarm manage \
```

```
            file://tmp/vagrant/swarm-cluster.cfg -H=0.0.0.0:1234
72acd5bc00de0b411f025ef6f297353a1869a3cc8c36d687e1f28a2d8f422a06
```

这个 Swarm 服务器配置使用了基于文件的发现机制。swarm-cluster.cfg 文件内容是硬编码的、由 Vagrant 启动的 Swarm 节点列表。除此之外，Swarm 还提供了其他服务发现方式（http://docs.docker.com/swarm/discovery/）。你可以使用 Docker 公司提供的发现服务、Consul（https://consul.io）、Etcd（https://github.com/coreos/etcd）或者 ZooKeeper（http://zookeeper.apache.org）等。你也可以编写自己的服务发现接口。

当 Swarm 服务器开始运行，工作节点都加入集群之后，你就可以使用本地 Docker 客户端来获取集群的信息或者启动容器。你需要在 Docker CLI 中使用 `-H` 参数来指定在容器中运行的 Swarm 服务器地址以代替本地的 Docker 守护进程，如下所示。

```
$ docker -H 0.0.0.0:1234 info
Containers: 0
Nodes: 3
 swarm-2: 192.168.33.12:2375
  └ Containers: 0
  └ Reserved CPUs: 0 / 1
  └ Reserved Memory: 0 B / 490 MiB
 swarm-3: 192.168.33.13:2375
  └ Containers: 0
  └ Reserved CPUs: 0 / 1
  └ Reserved Memory: 0 B / 490 MiB
 swarm-1: 192.168.33.11:2375
  └ Containers: 0
  └ Reserved CPUs: 0 / 1
  └ Reserved Memory: 0 B / 490 MiB
```

使用本地 Docker 客户端并指定 Swarm 服务器地址作为 Docker 守护进程端点，你可以在这个 Swarm 集群上运行容器。比如，让我们来启动一个 Nginx 容器，如下所示。

```
$ docker -H 0.0.0.0:1234 run -d -p 80:80 nginx
8399c544b61953fd5610b01be787cb3802e2e54f220673b94d78160d0ee35b33
$ docker -H 0.0.0.0:1234 run -d -p 80:80 nginx
1b2c4634fc6d9f2c3fd63dd48a2580f466590ddff7405f889ada885746db3cbd
$ docker -H 0.0.0.0:1234 ps
CONTAINER ID  IMAGE      ... PORTS                                NAMES
1b2c4634fc6d  nginx:1.7  ... 443/tcp, 192.168.33.11:80->80/tcp  swarm-1...
8399c544b619  nginx:1.7  ... 443/tcp, 192.168.33.12:80->80/tcp  swarm-2...
```

上面我们启动了两个 Nginx 容器。Swarm 将这两个容器调度到了集群中的两个节点之上。在浏览器中访问 http://192.168.33.11 和 http://192.168.33.12，就应该可以看到默认的 Nginx 页面了。

`docker run` 命令可能会花一些时间才能返回结果。Swarm 需要在集群中选择一个节点来运行容器，并且这个节点还需要去拉取 Nginx 镜像。

7.3.3 讨论

在这个例子中，Docker Swarm 服务器在 Swarm 头节点上的本地容器中运行。你可以通过 `docker ps` 命令看到这个容器，也可以通过 `docker logs` 命令查看这个容器的日志。在这个容器的日志中，你会看到启动 Nginx 容器的请求。这其实很有趣，你使用 Swarm 头节点上的 Docker 客户端来与该节点上的 Docker 守护进程通信，Swarm 服务器则在本地容器中运行，如下所示。

```
$ docker ps
CONTAINER ID  IMAGE         COMMAND      ...  PORTS
72acd5bc00de  swarm:latest  swarm manage ...  2375/tcp, 0.0.0.0:1234->1234/tcp

$ docker logs 72acd5bc00de
... msg="Listening for HTTP" addr="0.0.0.0:1234" proto=tcp
... msg="HTTP request received" method=GET uri="/v1.17/info"
... msg="HTTP request received" method=GET uri="/v1.17/containers/json"
... msg="HTTP request received" method=POST uri="/v1.17/containers/create"
... msg="HTTP request received" method=POST uri="/v1.17/containers/.../start"
... msg="HTTP request received" method=GET uri="/v1.17/containers/json"
... msg="HTTP request received" method=POST uri="/v1.17/containers/create"
... msg="HTTP request received" method=POST uri="/v1.17/containers/.../start"
... msg="HTTP request received" method=GET uri="/v1.17/containers/json"
```

在这些日志中，你会清楚地看到向 Swarm 服务器发送的、在集群中启动 Nginx 容器的 API 调用。

7.3.4 参考

- Swarm 简介（https://docs.docker.com/swarm/）
- Swarm 安装文档（https://docs.docker.com/swarm/install-manual/）

7.4 使用Docker Machine创建跨云计算服务提供商的Swarm集群

7.4.1 问题

你知道如何手动创建一个 Swarm 集群（参见范例 7.3），但是你想创建一个“节点分布在多个公共云计算服务提供商中”的 Swarm 集群，并且保持像在本地使用 Docker CLI 那样的用户体验。

7.4.2 解决方案

使用 Docker Machine（参见范例 1.9）在多个云计算服务提供商中启动 Docker 主机，并在对这些主机进行初始化时自动创建一个 Swarm 集群。

你需要做的第一件事情是获得一个 `swarm` 服务发现令牌。这个令牌会在启动集群中的节点

系统初始化时用到。如在范例 7.3 中已经介绍过的那样，Swarm 支持多种服务发现方式。在这个范例中，我们将使用 Docker 公司托管的服务发现方式。你可以通过创建一个基于 swarm 镜像的容器并执行 `create` 命令来获得一个用于服务发现的令牌。假设你还没有连接到一台 Docker 主机上，可以使用 `docker-machine` 创建一台专门用于获取服务发现令牌的 Docker 主机，如下所示。

```
$ ./docker-machine create -d virtualbox local
INFO[0000] Creating SSH key...
...
INFO[0042] To point your Docker client at it, run this in your shell: \
$(docker-machine env local)
$ eval "$(docker-machine env local)"
$ docker run swarm create
31e61710169a7d3568502b0e9fb09d66
```

获取令牌之后，你可以使用 `docker-machine` 命令在多个云计算服务提供商中启动 Swarm 集群的工作节点。你可以在 VirtualBox 上启动一个头节点，在 DigitalOcean 上（参见图 1-7）启动一个工作节点，在 Azure（参见范例 8.6）上启动另一个工作节点。

> 请不要在公有云中启动 Swarm 头节点，同时在本地的 VirtualBox 中启动工作节点。这样会带来头节点不能将网络通信路由到你本地工作节点的风险。尽管你也可以达到这个目的，但是这就需要在你的本地路由器上打开端口。

```
$ docker-machine create -d virtualbox --swarm --swarm-master \
--swarm-discovery token://31e61710169a7d3568502b0e9fb09d66 head
INFO[0000] Creating SSH key...
...
INFO[0069] To point your Docker client at it, run this in your shell: \
$(docker-machine env head)
$ docker-machine create -d digitalocean --swarm --swarm-discovery \
token://31e61710169a7d3568502b0e9fb09d66 worker-00
...
$ docker-machine create -d azure --swarm --swarm-discovery \
token://31e61710169a7d3568502b0e9fb09d66 swarm-worker-01
...
```

现在你的 Swarm 集群已经准备就绪。你的 Swarm 头节点在本地 VirtualBox 虚拟机中运行，有一个工作节点在 DigitalOcean 中运行，另一个工作节点则在 Azure 上运行。你可以将本地的 docker-machine 二进制文件设置为使用在 VirtualBox 中运行的头节点来运行 Swarm 子命令，如下所示。

```
$ eval "$(docker-machine env --swarm head)"
$ docker info
Containers: 4
Nodes: 3
 head: 192.168.99.103:2376
  └ Containers: 2
  └ Reserved CPUs: 0 / 4
  └ Reserved Memory: 0 B / 999.9 MiB
```

```
worker-00: 45.55.160.223:2376
  └ Containers: 1
  └ Reserved CPUs: 0 / 1
  └ Reserved Memory: 0 B / 490 MiB
swarm-worker-01: swarm-worker-01.cloudapp.net:2376
  └ Containers: 1
  └ Reserved CPUs: 0 / 1
  └ Reserved Memory: 0 B / 1.639 GiB
```

7.4.3 讨论

如果要启动一个新容器，Swarm 会以轮询方式在集群中进行调度。比如，你可以在一个 `for` 循环中启动三个 Nginx 容器，如下所示。

```
$ for i in`seq 1 3`;do docker run -d -p 80:80 nginx;done
```

这条命令会在集群的三个节点上，启动三个 Nginx 容器。记住，要想访问 Nginx 容器，需要打开你的公有云云主机实例的 80 端口（参见范例 8.6），如下所示。

```
$ docker ps
... IMAGE     ... PORTS                                  NAMES
... nginx:1.7 ... 443/tcp, 104.210.33.180:80->80/tcp    swarm-worker-01/
                                                         loving_torvalds
... nginx:1.7 ... 443/tcp, 45.55.160.223:80->80/tcp     worker-00/drunk_swartz
... nginx:1.7 ... 443/tcp, 192.168.99.103:80->80/tcp    head/condescending_galileo
```

别忘了删除你在云中启动的虚拟机。

7.4.4 参考

- 使用 Docker Machine 和 Docker Swarm（https://docs.docker.com/swarm/install-w-machine/）

7.5 使用Kitematic UI管理本地容器

7.5.1 问题

与使用命令行来管理本地容器相比，你更想使用图形界面工具对容器进行管理。

7.5.2 解决方案

使用 Kitematic（https://kitematic.com）。这个工具可以在 OS X 10.9+ 和 Windows 7+ 上使用。从 Docker 1.8 开始，Kitematic 已成为 Docker Toolbox 包的一个组件（参见范例 1.6）。

在通过从官方网站（https://kitematic.com）下载安装或者通过安装 Docker Toolbox 的方式安装完 Kitematic 之后，你就可以启动它了。它将会自动通过 Docker Machine 和 VirtualBox 来创建本地 Docker 主机。在这个 Docker 主机启动之后，你将会看到 Kitematic 的仪表盘（参见图 7-1）。

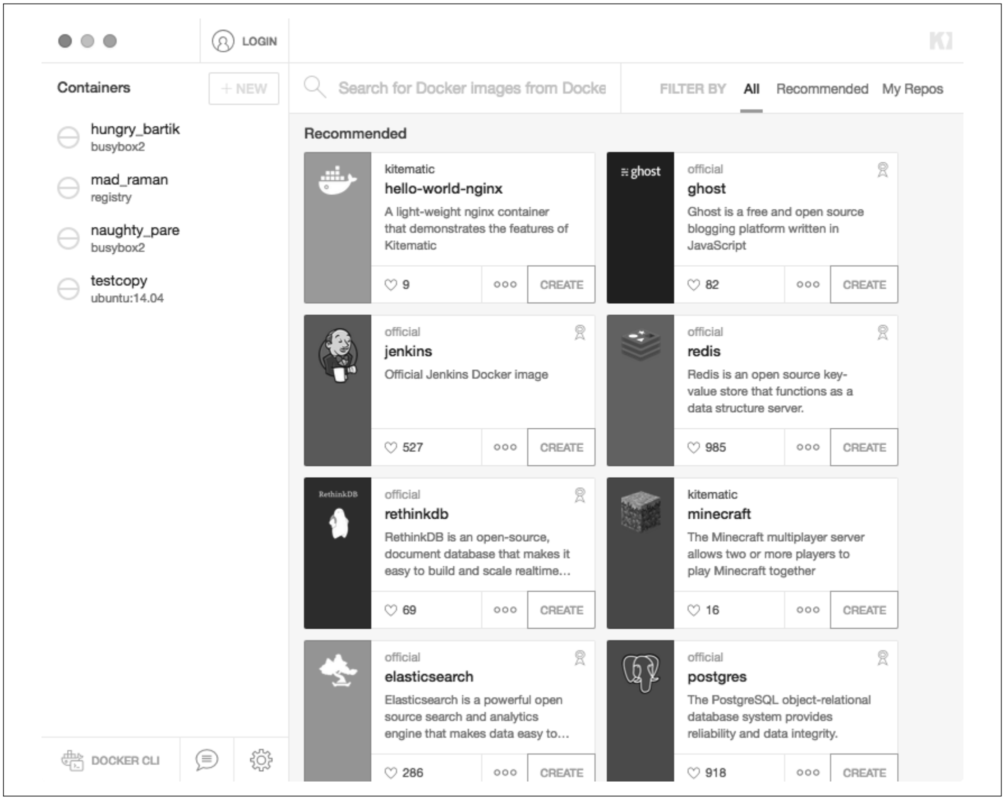


图 7-1：Kitematic 仪表盘

Kitematic 显示了默认的容器镜像，也允许你在 Docker Hub 上搜索。在图 7-1 中，左侧显示的是本地 Docker 主机中已有的容器。你可以通过 Kitematic UI 完成对这些容器的启动、停止以及管理容器配置的工作。

假设你想启动一个 Nginx 容器。可以在 Kitematic 仪表盘顶部的查找框中搜索 nginx 镜像，然后单击 Create 按钮。Docker 会自动下载镜像并启动容器。之后你可以进入设置窗口（参见图 7-2）来查看该容器的详细设置。你可以停止这个容器，修改容器的配置，然后重启该容器。

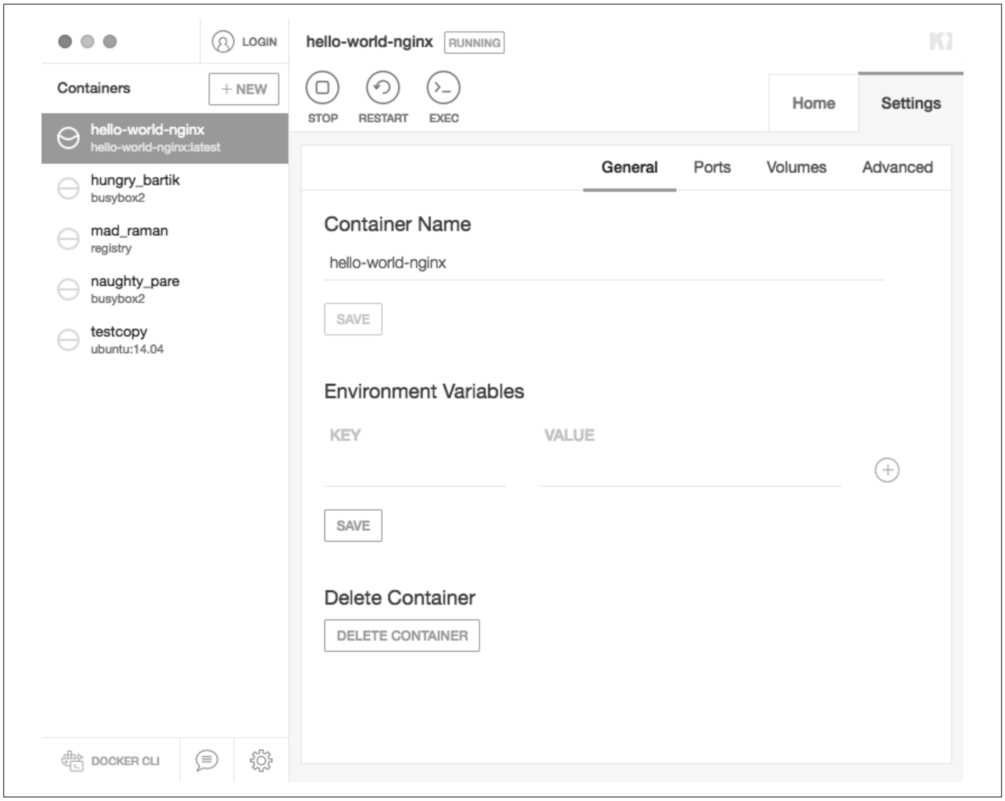


图 7-2：Kitematic 设置窗口

使用 Kitematic 可以非常方便地进行 Docker 的本地工作和简单的容器管理任务。Kitematic 将来很可能会被集成到 Docker Compose 和 Swarm 中，以提供一个用于管理生产环境部署的强大图形界面工具。

7.5.3 参考

- Kitematic 官方网站（https://kitematic.com）

7.6 使用Docker UI管理容器

7.6.1 问题

你已经可以登录到一台 Docker 主机，知道如何管理镜像和容器，但是你想使用一个简单的 Web 界面来进行管理。

7.6.2 解决方案

使用 Docker UI（https://github.com/crosbymichael/dockerui）。尽管你可以从源代码构建自

己的 Docker UI 镜像（https://github.com/crosbymichael/dockerui/wiki/Ways-to-run-dockerui），但是也可以直接使用 Docker Hub（https://registry.hub.docker.com/u/dockerui/dockerui/）上的 Docker UI 镜像，这样更方便在容器中运行 Docker UI。

在你的 Docker 主机上，启动 Docker UI 容器，如下所示。

```
$ docker run -d -p 9000:9000
                --privileged
                -v /var/run/docker.sock:/var/run/docker.sock
                dockerui/dockerui
```

然后，你可以打开浏览器访问 http://<IP_OF_DOCKER_HOST>:9000，就会看到 Docker UI。图 7-3 是 Docker UI 的一个例子。

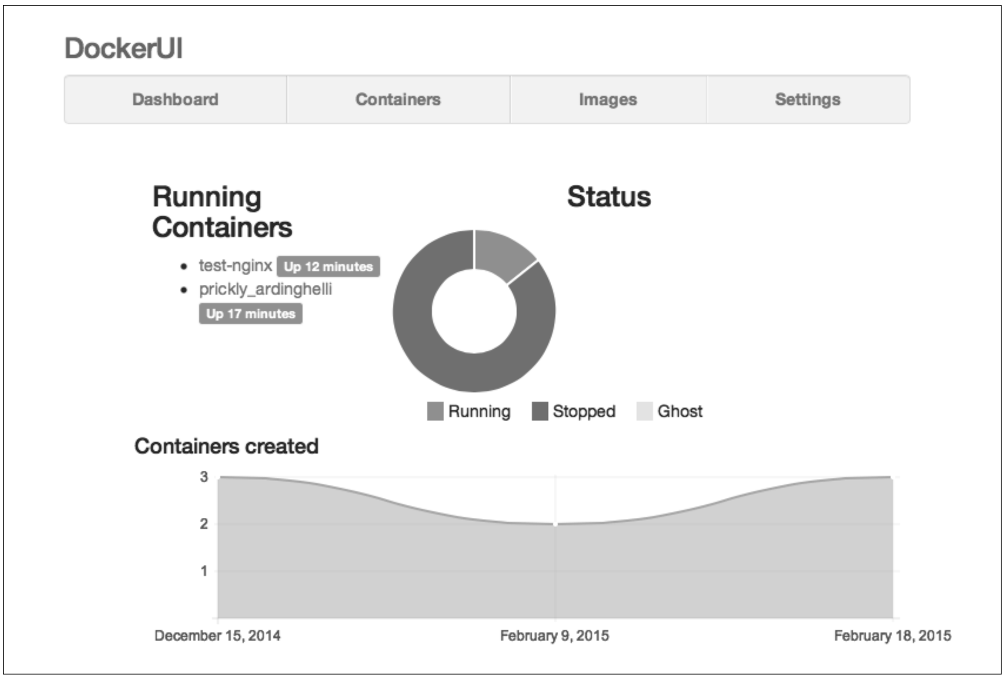
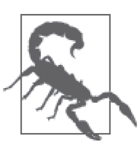


图 7-3：Docker UI 仪表盘



Docker UI 并非由官方 Docker 发布的，而是一个由社区维护的项目（https://github.com/crosbymichael/dockerui）。

7.6.3　讨论

打开 Docker UI 之后，你就可以启动容器了。进入到 Images 选项卡，选择一个镜像，然后单击 Create 按钮。之后会弹出用于设置所有容器启动选项的 UI。不要忘了 HostConfig 选项，你可以在这里进行端口映射设置。图 7-4 是 Docker UI 的预览。

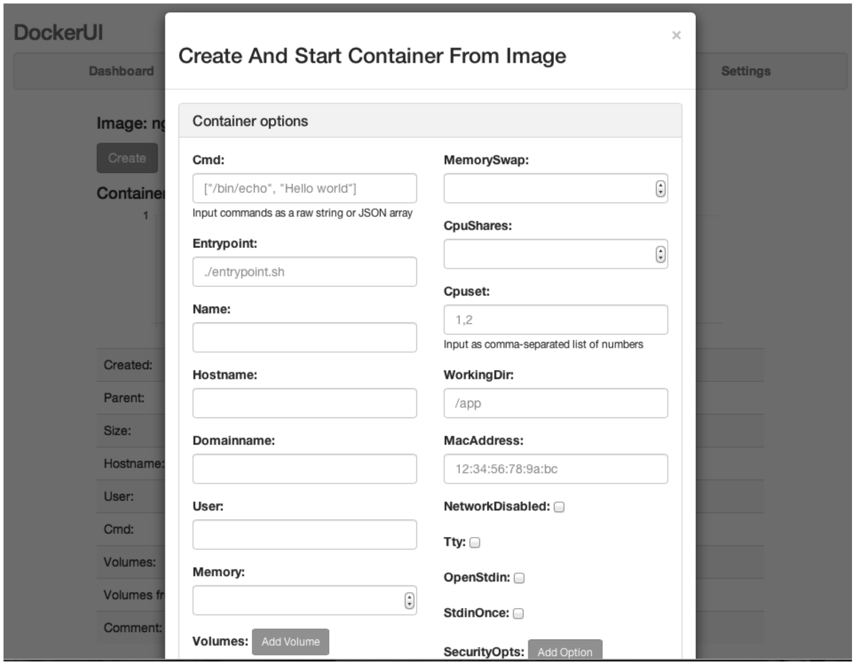


图 7-4：通过 Docker UI 启动一个容器

7.6.4 参考

- Docker UI 的 wiki 中有更多信息（https://github.com/crosbymichael/dockerui/wiki）

7.7 使用Wharfee交互式shell

7.7.1 问题

你知道如何使用 Docker CLI，但是你想使用一个具备自动补全和历史记录功能的更强大的交互式 shell。

7.7.2 解决方案

使用 Wharfee（http://wharfee.com），这是一个由 Python 编写的交互式 CLI。Wharfee 使用 docker-py（参见范例 4.10）和其他一些 Python 模块实现了一个交互式 shell。

在一台 Docker 主机上，安装 Python 和包管理命令行工具 `pip`。然后安装 Wharfee。比如在 Ubuntu 系统上，可以像下面这样操作。

```
$ sudo apt-get install python python-pip
$ sudo pip install wharfee
```

现在你就可以使用这个新的强大的命令行工具了。启动 Wharfee 后，你会进入到交互式 shell。在这个 shell 里，你可以运行的命令与 Docker 命令完全一样，只不过不用再使用 docker 前缀了。在输入的同时，你也会看到自动补全提示和一些语法高亮。

```
$ wharfee
Version: 0.6.5
Home: http://wharfee.com
wharfee> images
There are no images to list.
wharfee> ps
There are no containers to list.
```

要想启动一个容器，需要先拉取指定的镜像，如下所示。

```
wharfee> pull nginx:latest
Pulling from library/nginx latest
...
wharfee> run -d -p 80:80 nginx:latest
bf96488c76d617b6d3d2f8aea0ff928eff7fe05e61219eb23f865f60631d9f83
wharfee> ps
Status        Created  Image         ... Command                  Ports
------------  -------  ------------  ... ----------------------   ---------------
Up 2 seconds  now      nginx:latest  ... nginx -g 'daemon off;'   443/tcp, 80/tcp
```

Wharfee 是一个很不错的 CLI 工具，具有自动补全和语法高亮功能，使用起来非常方便。

7.7.3 参考

- Wharfee 源代码（https://github.com/j-bennet/wharfee）
- Wharfee 官方网站（http://wharfee.com）

7.8 使用Ansible的Docker模块对容器进行编排

7.8.1 问题

你已经积累了一些使用 Ansible（http://www.ansible.com/home）进行服务器配置和对应用程序部署进行编排的经验。现在，你希望能充分利用这些经验，使用 Ansible 来管理 Docker 容器。

7.8.2 解决方案

使用 Ansible 的 Docker 模块（http://docs.ansible.com/docker_module.html）。这个模块是 Ansible 核心的一部分，因此在安装完 Ansible 之后，你不需要再安装额外的软件包。

Ansible 将会在你的本地主机上运行，通过 SSH 连接到远程 Docker 主机，使用 docker-py API 客户端对 Docker 守护进程发起请求。

比如，要想以后台方式启动一个 Nginx 容器并设置端口映射，可以使用下面的 Ansible playbook（http://docs.ansible.com/playbooks.html）。

```
- hosts: nginx
  tasks:
  - name: Run nginx container
    docker: image=nginx:latest detach=true ports=80:80
```

讨论如何使用 Ansible 已经超出了本范例的范畴，你可以参考一下 Ansible 的文档（http://docs.ansible.com）。

7.8.3 讨论

为了帮助你快速学会使用 Ansible 的 Docker 模块，你可以使用本书范例附带的 Vagrantfile 文件。这个 Vagrantfile 将会启动一个虚拟机作为 Docker 主机，里面安装了 `docker-py` 客户端。此外还为你准备了两个 playbook、一个 inventory 文件以及一些 Ansible 的配置文件，你可以直接拿来使用。

第一个任务是需要在你的本地主机上安装 Ansible，如下所示。

```
$ sudo pip install ansible
```

然后测试 Nginx playbook，可以按照下面的命令操作。

```
$ git clone https://github.com/how2dock/docbook.git
$ cd ch07/ansible
$ tree
.
├── README.md
├── Vagrantfile
├── ansible.cfg
├── dock.yml
├── inventory
├── solo
│   ├── Vagrantfile
│   └── dock.yml
└── wordpress.yml
$ vagrant up
```

解决方案部分中提到的 Nginx playbook 在 dock.yml 文件中。要想使用 Ansible 启动这个容器，你可以执行这个 playbook。当这个 playbook 执行完毕之后，打开你的浏览器访问 http://192.168.33.10，就能看到 Nginx 的欢迎页面。你也可以通过 `vagrant ssh` 登录到该虚拟机，然后使用通常的 `docker ps` 命令来查看正在运行中的容器。

```
$ ansible-playbook -u vagrant dock.yml
PLAY [nginx] ******************************************************************

GATHERING FACTS ***************************************************************
ok: [192.168.33.10]

TASK: [Run nginx container] ***************************************************
changed: [192.168.33.10]
```

```
PLAY RECAP ********************************************************************
192.168.33.10              : ok=2    changed=1    unreachable=0    failed=0
```

你可以在虚拟机中通过 `docker kill` 命令终止容器，或者运行一个 playbook 来将容器的状态设置为 `killed`，如下所示。

```
- hosts: nginx
  tasks:
  - name: Kill nginx container
    docker: image=nginx:latest detach=true ports=80:80 state=killed
```

如果你想尝试一个更复杂的例子，可以使用 WordPress 的 playbook wordpress.yml。你已经部署过多次 WordPress 了（参考范例 1.15 或范例 1.16）。运行这个 playbook，然后打开浏览器访问 http://192.168.33.10，你就可以再次享受 WordPress 了。（在这之前需要先终止占用了 80 端口的容器，否则会出现端口冲突错误。）

```
$ ansible-playbook -u vagrant wordpress.yml

PLAY [wordpress] **************************************************************

GATHERING FACTS ***************************************************************
ok: [192.168.33.10]

TASK: [Docker pull mysql] *****************************************************
changed: [192.168.33.10]

TASK: [Docker pull wordpress] *************************************************
changed: [192.168.33.10]

TASK: [Run mysql container] ***************************************************
ok: [192.168.33.10]

TASK: [Run wordpress container] ***********************************************
changed: [192.168.33.10]

PLAY RECAP ********************************************************************
192.168.33.10              : ok=5    changed=3    unreachable=0    failed=0
```

由于 Ansible 的 playbook 都是 YAML 格式的，你会注意到上面的 playbook 文件和范例 7.1 中 WordPress 的 Docker Compose 文件很像，如下所示。

```
- hosts: wordpress
  tasks:

  - name: Docker pull mysql
    command: docker pull mysql:latest

  - name: Docker pull wordpress
    command: docker pull wordpress:latest

  - name: Run mysql container
    docker:
      name=mysql
      image=mysql
```

```
      detach=true
      env="MYSQL_ROOT_PASSWORD=wordpressdocker,MYSQL_DATABASE=wordpress, \
          MYSQL_USER=wordpress,MYSQL_PASSWORD=wordpresspwd"

  - name: Run wordpress container
    docker:
      image=wordpress
      env="WORDPRESS_DB_NAME=wordpress,WORDPRESS_DB_USER=wordpress, \
           WORDPRESS_DB_PASSWORD=wordpresspwd"
      ports="80:80"
      detach=true
      links="mysql:mysql"
```

> 你已经在本地主机上直接运行了 playbook，不过 Vagrant 提供了一个 Ansible 配置程序。也就是说，你可以在 VM 启动时运行 playbook。进入到 ch07/ansible/solo 文件夹，然后运行 `vagrant up`，Nginx 容器也会随着 VM 自动启动。

7.8.4　参考

- *Ansible: Up and Running* 中关于 Docker 模块的章节（http://shop.oreilly.com/product/0636920035626.do）

7.9　在Docker主机集群中使用Rancher管理容器

7.9.1　问题

你需要在生产环境中使用能支持多主机网络的系统来管理容器，一个覆盖网络允许容器可以不通过复杂的端口映射配置就可以互相通信。你也希望这个系统支持分组管理，而且还能提供一个功能强大的仪表盘。

7.9.2　解决方案

可以考虑使用来自 Rancher Labs（http://rancher.com）的 Rancher（http://rancher.com/rancher-io/），Rancher Labs 也是 RancherOS 的开发者（参见范例 6.10 ）。它安装起来比较简单，你需要一个以容器方式运行的管理服务器，以及一个同样以容器方式运行的工作代理。

为了方便对 Rancher 进行测试，以及确认一下 Rancher 是否能满足你的需求，你可以从 GitHub（https://github.com/rancherio/rancher）克隆 Rancher 项目的仓库，通过 Vagrant 在本地启动一个虚拟机，如下所示。

```
$ git clone https://github.com/rancherio/rancher.git
$ cd rancher
$ vagrant up
```

上面命令启动的虚拟机基于 CoreOS（参见范例 6.1），不过你也可以使用任何能运行 Docker 的操作系统。这个 Vagrantfile 文件包括两个初始化的配置步骤，一个用于安装管理服务器，一个用于安装工作代理。这两个步骤都使用了 Docker 镜像来运行相应的软件。

你也可以在自己的计算机上使用这些命令来运行 Rancher。

在 Rancher 仪表盘页面，如果单击 Add Host 按钮，你将会看到为了将新主机加入到 Rancher 部署中，你需要在其他主机上运行的完整的 Docker 命令。

```
$ docker run -d -p 8080:8080 rancher/server:latest
$ docker run -e CATTLE_AGENT_IP=172.17.8.100 --privileged -e WAIT=true \
        -v /var/run/docker.sock:/var/run/docker.sock \
    rancher/agent:latest http://localhost:8080
```

当 Vagrant 主机启动之后，Rancher 镜像将会被下载，同时会启动两个容器，之后你就可以通过 http://localhost:8080 访问 Rancher 仪表盘。

如果你本地主机上的 8080 端口已经被占用，那么 Vagrant 将会选择其他端口来运行 Rancher UI。不过你始终可以通过使用主机网络来访问 Rancher UI，网址为 http://172.17.8.100:8080。

这个仪表盘将只会显示一台主机，并且没有运行中的容器。单击 Add Container，进入设置容器启动参数的页面（参见图 7-5 ）。你可以展开 Advanced Options 区域来设置诸如环境变量、卷、网络和容器的内核能力（比如内存、特权模式等）。默认情况下，容器的网络模式为托管网络，这种模式会使用一个覆盖网络。不过你也可以使用默认的 Docker 网络模式。

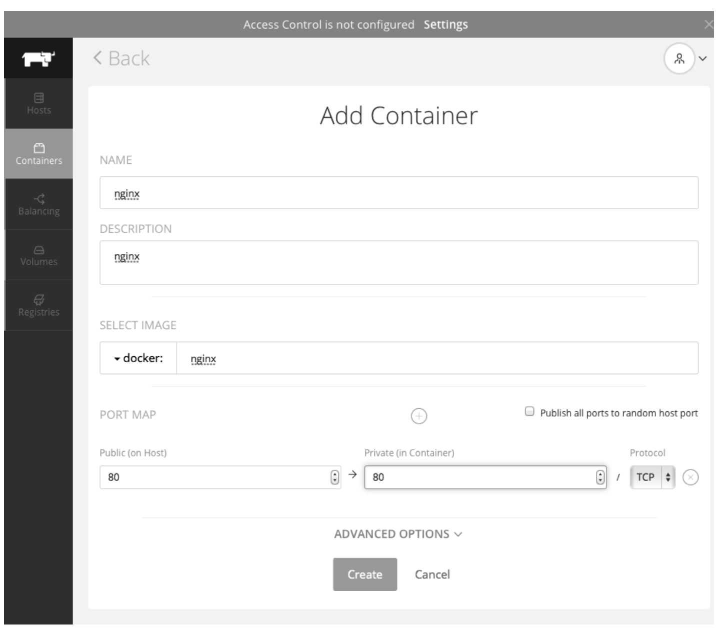


图 7-5：通过 Rancher UI 启动容器

考虑到 Docker 的网络实现还可能会有变动，因此这个范例并不会对 Rancher 的覆盖网络本身展开讲解。可以参考第 3 章获得更多信息。

Rancher 将会构建一个覆盖网络，尽管我们这个例子是在一台主机上。Rancher 会在覆盖网络的 IP 地址范围内启动容器。如果你将容器的端口映射到宿主机的端口上，就可以通过浏览器访问该容器。举例来说，如果你启动一个 Nginx 容器并将它的端口映射到宿主机的 80 端口上，就可以访问到 Nginx 的欢迎页面。创建完成的容器界面如图 7-6 所示。

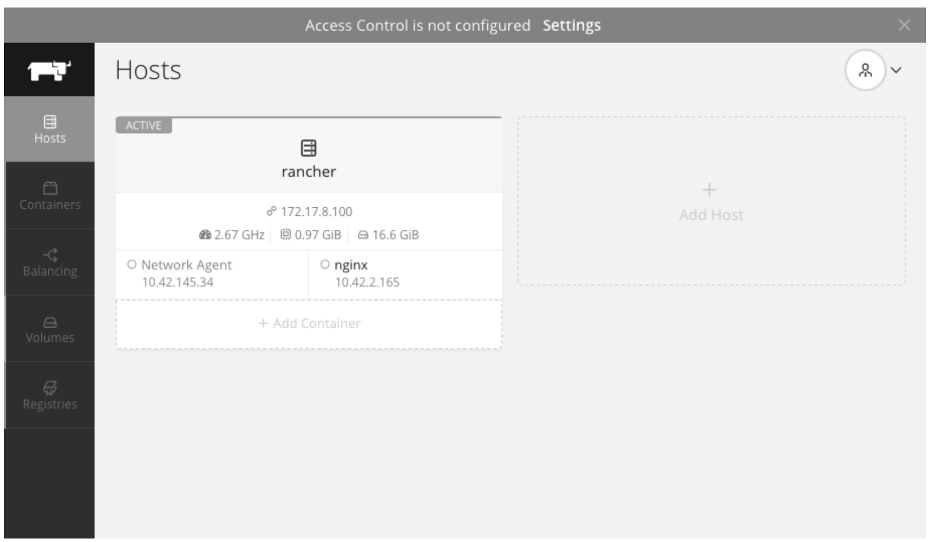


图 7-6：显示了运行中容器的 Rancher 仪表盘

到目前为止，你已经拥有了一个可以进行测试的 Rancher 环境。你可以浏览一下 Rancher 仪表盘。Containers 选项卡显示了所有运行中的容器。你可以打开一个链接到容器的 shell，并启动和停止容器。Volumes 选项卡显示了当前使用的所有卷，当前能通过仪表盘对卷进行的处理也很有限。最后，你可以定义一个现有的私有 registry，或者定义一个负载平衡器。

7.9.3 讨论

到这里，你的所有操作都是通过 Rancher 仪表盘来完成的。实际上 Rancher 还提供了一套 REST API 来管理它的所有资源。要想使用这些 API，你需要先创建一组 API 访问标识和密钥。你可以通过单击仪表盘右上角的 User 图标然后选择 API & Keys 选项。尽管在 GitHub 的页面上并没有关于这些 API 的详细说明，不过在 Rancher 仪表盘中你可以方便地浏览这些 API。

你可以通过仪表盘来管理一个运行中的容器。单击这个容器，你会看到用于查看 API 的选项。单击这个选项就可以查看这个 API 的详情。这个浏览 API 的功能会以 JSON 对象的格式显示容器的描述信息以及一些可以对该容器采取的操作（UI 中的绿框）。选择其中的一个操作会打开一个新窗口，显示你可以使用的 API 请求。这是一个非常好的学习 Rancher API 的方式，你也可以据此来编写自己的 Rancher 客户端。图 7-7 是一个用于停止容器的请求示例。



图 7-7：Rancher API 请求示例

7.10 使用Lattice在集群中运行容器

7.10.1 问题

你在寻找一个对容器进行编排的系统，在集群中的计算机之间进行容器调度。此外，你有一些 Cloud Foundry 的经验，并且对 Cloud Foundry 管理容器的方式感兴趣。

7.10.2 解决方案

使用 Lattice（http://lattice.cf）。Lattice 是一个面向基于容器的应用程序的集群调度器，它包括一个 HTTP 负载平衡器、日志聚合、健康管理和应用程序的动态扩容、缩容的功能。这个轻量级的调度器为开发人员带来了原生云应用和 PaaS 平台 Cloud Foundry（https://www.cloudfoundry.org）的体验。

为了快速入门，你可以按照 Lattice 官方网站（http://lattice.cf/docs/getting-started/）的入门

指南进行操作。这个入门指南使用了一个 Vagrant 虚拟机在你的本地计算机上部署一个 Lattice cell。在安装完最新版的 Lattice 客户端程序 `ltc` 之后，就可以与你的 Lattice 服务进行通信并部署 Docker 镜像了。

让我们克隆 Lattice 项目，切换到最新的发布版本，然后启动 Vagrant 虚拟机，如下所示。

```
$ git clone https://github.com/cloudfoundry-incubator/lattice.git
$ cd lattice
$ git checkout v0.3.0
$ vagrant up
...
==> default: Lattice is now installed and running.
==> default: You may target it using: ltc target 192.168.11.11.xip.io
==> default:
```

下载 `ltc` 命令行工具，并设置为可执行权限。比如，为了将该工具添加到你的 `PATH` 中，可以使用下面的命令。

```
$ sudo wget https://lattice.s3.amazonaws.com/releases/latest/darwin-amd64/ltc \
-O /usr/local/bin/ltc
$ sudo chmod +x /usr/local/bin/ltc
```

现在就可以配置这个命令行工具，让它指向你在 Vagrant 中部署的本地 Lattice 服务了。按照 Vagrant 启动时输出的消息操作，并使用 `ltc` 工具，如下所示。

```
$ ltc target 192.168.11.11.xip.io
Blob store is targeted.
Api Location Set
```

现在剩下的工作就是通过 `ltc create` 命令来启动容器了，指定一个你想要运行的 Docker Hub 中的 Docker 镜像。作为一个简单的测试，让我们在 Lattice 中启动一个 `nginx` 容器，如下所示。

```
$ ltc create nginx-app nginx -r
...
nginx-app is now running.
App is reachable at:
http://nginx-app.192.168.11.11.xip.io
```

当应用程序创建完成之后，你可以通过 `ltc` 命令返回的 URL 来访问这个应用。这个 URL 使用了 xip.io（http://xip.io）泛域名服务。通过该 DNS 服务，你可以更容易地通过 DNS 名来访问本地网络上的服务。

在这个例子中，如果你在浏览器中打开 http://nginx-app.192.168.11.11.xip.io，就会看到 Nginx 的欢迎页面。

你可以使用 `ltc scale` 命令很轻松地调整一个应用程序实例的数量，使用 `ltc rm` 命令来停止一个应用程序，以及很多其他的操作任务。运行 `ltc help` 可以获得更多帮助信息。

7.10.3 讨论

通过 Vagrant 启动的 Lattice cell 并没有运行 Docker 引擎。但是你在运行时使用了 Docker

镜像并且 Lattice 能够正常工作。实际上，Lattice 运行时先是将 Docker 镜像的文件系统展开，然后在自己的容器运行时中运行该应用程序。

这个例子中的做法其实有一个副作用：为了 Nginx 能部署成功，你需要指定以 root 身份来运行应用。这是通过 `ltc create` 命令的 `-r` 选项来实现的。

尽管这个基本的示例使用了 Vagrant，但实际上使用 Terraform（https://github.com/cloudfoundry-incubator/lattice/tree/master/terraform）Lattice 能够在很多公有云上部署。

7.10.4 参考

- 在 Lattice 中使用 buildpacks（http://www.chipchilders.com/blog/2015/8/12/buildpacks-in-lattice.html）

7.11 通过Apache Mesos和Marathon运行容器

7.11.1 问题

你在寻找一个集群调度器，用于在你数据中心的 Docker 主机上启动容器。你也许已经尝试过了在 Apache Mesos 上调度长时间运行的任务、定时任务，甚至 Hadoop 或者并行计算处理，现在你想在 Mesos 上运行容器。

7.11.2 解决方案

使用 Apache Mesos（http://mesos.apache.org）和 Docker 容器化（containerizer）。Mesos 是一个利用多种调度框架来最大化你的数据中心资源利用率的集群资源分配工具。很多大公司，包括 eBay、Twitter、Netflix、Airbnb 等（http://mesos.apache.org/documentation/latest/powered-by-mesos/），都在使用 Mesos。

Mesos 架构（http://mesos.apache.org/documentation/latest/mesos-architecture/）包括一台或者多台主节点以及工作节点（或被称为 Mesos 从属节点），一个或者多个在 Mesos 集群上部署的调度框架，以及一个使用了 ZooKeeper（http://zookpeeper.apache.org）的服务发现系统。在范例 7.2 中，你已经了解了如何使用 Docker Compose 在单节点上启动一个测试用的 Mesos 基础设施。

Marathon（http://mesos.apache.org/documentation/latest/mesos-frameworks/）是一个允许你在 Mesos 集群中运行任务的 Mesos 框架。Mesos 支持 Docker（即 Docker 容器化）。也就是说，你可以启动由 Docker 容器构成的 Marathon 任务。

> Amazon ECS 服务（参见范例 8.11）也可以使用 Mesos 在 AWS 上对容器进行调度。Docker Swarm（参见范例 7.3）也计划增加对基于 Mesos 的调度的支持。

这个范例将要使用 Mesos Playa（https://github.com/mesosphere/playa-mesos），它是一个 Mesos 沙箱，我们会基于 Playa 对如何在 Mesos 中运行 Docker 容器进行说明。

作为开始，先从 GitHub 上克隆 `playa-mesos` 这个仓库，通过 Vagrant 启动虚拟机，然后 `ssh` 到虚拟机中，如下所示。

```
$ git clone https://github.com/mesosphere/playa-mesos.git
$ vagrant up
$ vagrant ssh
```

虚拟机启动之后，你可以通过 http://10.141.141.10:5050 访问 Mesos Web 界面，以及通过 http://10.141.141.10:8080 访问 Marathon UI。

> 讨论如何使用 Mesos（http://mesos.apache.org）和 Marathon（http://https://mesosphere.github.io/marathon/）已超出了本书范围，可以访问这两个网站来获得更多信息。

你可以通过 Marathon 提供的一套 REST API 来启动任务。Marathon 的任务以 JSON 文件的格式来定义，并通过 `curl` 命令提交到 API 端点。下面是一个使用 JSON 描述的 Marathon 示例任务。

```
{
  "id": "http",
  "cmd": "python -m SimpleHTTPServer $PORT0",
  "mem": 50,
  "cpus": 0.1,
  "instances": 1,
  "constraints": [
    ["hostname", "UNIQUE"]
  ],
  "ports": [0]
}
```

其中 `id` 是任务的名称（在 Marathon 中也被称为应用程序）。`cmd` 表明了应用程序将要运行的命令（这里是一个使用 Python 编写的简易 HTTP 服务器）。值得重点一提的是 `ports` 的使用，这里它被设置为一个只包含一个 `0` 的列表。这样设置表示 Marathon 会自动为这个应用程序分配一个端口号。这个动态端口号会以变量 `$PORT0` 的形式传给 `cmd`。

将上面的 JSON 定义保存到文件并命名为 test.json，然后通过 `curl` 命令来提交这个应用程序。

```
$ curl -is -H "Content-Type: application/json"
           -d @test.json 10.141.141.10:8080/v2/apps
HTTP/1.1 201 Created
...
```

当应用程序启动之后，你将会在 UI 中看到这个应用（参见图 7-8），也可以通过指向刚刚启动的 HTTP 服务器的 URL 来访问该应用程序。注意，端口号是动态分配的。

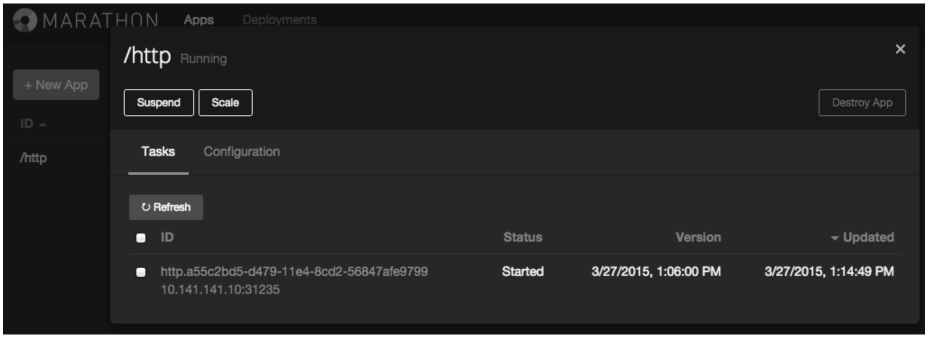


图 7-8：Marathon UI 中的 HTTP 服务

下面让我们来启动一个由 Docker 容器组成的应用程序。默认情况下，Playa Mesos 启动的虚拟机里面已经安装了 Docker，但是 Mesos 从属节点并没有配置好来使用 Docker。因此，你需要进行一点修改，然后重启 `mesos-slave`。在虚拟机中执行如下命令。

```
vagrant@mesos:~$ sudo su
root@mesos:/home/vagrant# cd /etc/mesos-slave
root@mesos:/etc/mesos-slave# echo 'docker,mesos' > containerizers
root@mesos:/etc/mesos-salve# echo '5mins' > executor_registration_timeout
root@mesos:/etc/mesos-slave# service mesos-slave restart
mesos-slave stop/waiting
mesos-slave start/running, process 2581
```

创建下面的 JSON 文件（比如 docker.json）来定义一个用于启动 Nginx 容器并使用 Docker 主机动态端口分配的应用。

```
{
  "container": {
    "type": "DOCKER",
    "docker": {
      "image": "nginx",
      "network": "BRIDGE",
      "portMappings": [
        { "containerPort": 80, "hostPort": 0 }
      ]
    }
  },
  "id": "nginx",
  "instances": 1,
  "cpus": 0.5,
  "mem": 512
}
```

运行 `curl` 命令，通过 Marathon API 创建这个应用程序，然后查看一下运行中的应用程序，如下所示。

```
$ curl -si -H 'Content-Type: application/json'
           -d @docker.json 10.141.141.10:8080/v2/apps
```

```
$ curl -sX GET -H "Content-Type: application/json" 10.141.141.10:8080/v2/tasks
          | python -m json.tool
{
   "tasks": [
       {
           "appId": "/nginx",
           "host": "10.141.141.10",
           "id": "nginx.404b7376-d47b-11e4-8cd2-56847afe9799",
           "ports": [
               31236
           ],
           "servicePorts": [
               10001
           ],
           "stagedAt": "2015-03-27T12:17:35.285Z",
           "startedAt": null,
           "version": "2015-03-27T12:17:29.312Z"
       },
       {
           "appId": "/http",
           "host": "10.141.141.10",
           "id": "http.a55c2bd5-d479-11e4-8cd2-56847afe9799",
           "ports": [
               31235
           ],
           "servicePorts": [
               10000
           ],
           "stagedAt": "2015-03-27T12:06:05.873Z",
           "startedAt": "2015-03-27T12:14:49.986Z",
           "version": "2015-03-27T12:06:00.485Z"
       }
   ]
}
```

在任务列表中，你看到了之前启动的 `http` 应用程序。你也看到了新启动的使用了 Docker nginx 镜像的应用。这个应用程序需要多花费一些时间进行部署，因为它需要花时间去执行 `docker pull nginx` 操作。考虑到从 registry 下载镜像所需的时间，你在重启 `mesos-slave` 之前设置了 `executor_registration_timeout` 参数。Marathon 也将 Nginx 容器的 80 端口动态地映射到了宿主机上，在这个例子中它选择了 31236。如果在浏览器中打开 http://10.141.141.10:31236，你就能看到熟悉的 Nginx 欢迎页。

7.11.3 讨论

JSON 格式文件中的 Docker 应用程序定义可以设置很多属性，包括卷挂载，指定参数覆盖在 Dockerfile 中 CMD 指令设置的参数，可以指定 `docker run` 的参数，也可以在特权模式中运行容器。Docker 容器化（https://mesosphere.github.io/marathon/docs/native-docker.html）文档提供了详细的文档。但作为快速参考，你也可以定义一个使用了所有这些额外功能的应用程序，如下所示。

```
{
    "id": "privileged-job",
    "container": {
        "docker": {
            "image": "mesosphere/inky"
            "privileged": true,
            "parameters": [
                { "key": "hostname", "value": "a.corp.org" },
                { "key": "volumes-from", "value": "another-container" },
                { "key": "lxc-conf", "value": "..." }
            ]
        },
        "type": "DOCKER",
        "volumes": []
    },
    "args": ["hello"],
    "cpus": 0.2,
    "mem": 32.0,
    "instances": 1
}
```

最后，在单台主机上运行 Mesos 违背了本范例的初衷，并且你也希望创建一个在所有从属节点上使用 Docker 容器化的 Mesos 集群。

7.11.4 参考

- Mesosphere 关于 Docker 容器的文档（https://mesosphere.github.io/marathon/docs/native-docker.html）
- Marathon JSON 示例文件（https://github.com/mesosphere/marathon/tree/master/examples）
- 本范例参考的博客文章（http://frankhinek.com/deploy-docker-containers-on-mesos-0-20/）

7.12 在Mesos集群上使用Mesos Docker容器化

7.12.1 问题

在范例 7.11 中，你看过了如何对 Mesos Docker 容器化进行测试并在 Mesos 沙箱中运行容器。你希望在 Mesos 集群中完成同样的工作。

7.12.2 解决方案

使用 Docker Hub 上 Mesosphere（https://mesosphere.com）提供的镜像来构建一个在容器中运行的 Mesos 集群。配置 Mesos 从属节点使用 Docker 容器化。

为了完成本范例中的测试，你可以使用本书附带的仓库内容。本范例将通过一个 Vagrantfile 文件安装一个由三个本地节点构成的 Mesos 集群，并使用 Ansible 来启动运行 Mesos 软件组件（即 ZooKeeper、Mesos 主节点、Marathon 框架和 Mesos 从属节点）的容器。

如果还没有克隆这个仓库，你需要克隆一下这个仓库，进入到 ch07/mesos 文件夹下，然后

通过 Vagrant 启动这个三节点的集群。

> 如果你的宿主机有足够多的内存，也可以为这个集群增加更多的节点，或者增加为每个节点分配的内存（参考 Vagrantfile 文件）。

```
$ git clone https://github.com/how2dock/docbook.git
$ cd dockbook/ch07/mesos
$ vagrant up
$ vagrant status
Current machine states:

mesos-head                running (virtualbox)
mesos-1                   running (virtualbox)
mesos-2                   running (virtualbox)
```

如果你是按照范例的顺序来阅读本书的，那么应该已经阅读过范例 7.8。如果你还没有阅读这个范例，那么请先阅读这个范例，在你的宿主机上对 Ansible（http://ansible.com）进行设置。我们将要使用一个 Ansible playbook 在虚拟机中启动一些容器。这个 playbook 的文件名为 mesos.yml。要想启动所有容器，你可以运行这个 playbook，如下所示。

```
$ ansible-playbook -u vagrant mesos.yml
```

当上面的命令执行完毕，Mesos 头节点上将会有三个容器在运行（即 ZooKeeper、Mesos 主节点和 Marathon 框架）。两个从属节点上都会有一个容器在运行（即 Mesos 从属节点）。所有这些镜像都来自 Docker Hub。

打开你的浏览器访问 http://192.168.33.10:5050 就可以看到 Mesos UI，访问 http://192.168.33.10:8080 则可以看到 Marathon UI。

要想在 Mesos 集群中启动一个 Nginx 容器，可以通过 API 在 Marathon 框架中创建一个 Mesos 应用程序，如下所示。

```
$ curl -si -H 'Content-Type: application/json' \
         -d @docker.json 192.168.33.10:8080/v2/apps
```

当 Nginx 镜像下载完成之后，就能访问 Nginx 的欢迎页面了，这与范例 7.11 中介绍的内容非常类似。

> 应用程序定义文件 docker.json 指定了 128 MB 的内存。如果你的从属节点没有足够多的内存，则应用程序可能会卡在部署阶段。确保你的从属节点有足够多的内存，以减少对应用程序的限制。

7.12.3　讨论

Ansible 使用的清单被硬编码到了 inventory 文件。如果你修改了节点的 IP 地址或者增加了节点，请确保同时更新清单。

目前 Ansible playbook 通过 SSH 远程执行 `docker run` 命令。如果你想使用 Ansible 的 Docker 模块，请注释掉 `command` 任务，然后删除对 `docker` 任务的注释。

你也许已经注意到了，Mesos 从属节点是以容器的形式在运行。当启动这个容器时，你设置了 `MESOS_CONTAINERIZERS=docker,mesos` 环境变量，用于告诉 Mesos 从属节点使用 Docker。从属节点将会在其宿主机上启动其他的容器。这通过将宿主机上的 /var/run/docker.sock、/usr/bin/docker 和 /sys 挂载到容器中来实现。尽管在测试环境下这种方式能正常工作，但是 Mesos Docker 容器并不是用来做这件事的。Mesos 建议在生产环境下在容器中运行从属节点之前，你应该让 Mesos 从属节点在宿主机上运行。

7.12.4 参考

- Apache Mesos 配置说明（http://mesos.apache.org/documentation/latest/configuration/）

7.13 使用registrator发现Docker服务

7.13.1 问题

你正在构建一个分布式应用程序，该程序的服务在跨越多台主机的容器中运行。你需要能自动发现这些服务来对应用程序进行配置。在一个服务从一台主机迁移到另一台主机的情况下，或者在服务自动启动的情况下，需要对应用程序进行配置。

7.13.2 解决方案

使用 registrator（https://github.com/gliderlabs/registrator）。registrator 以容器的形式在你的系统中运行。通过将 Docker socket /var/run/docker.sock 挂载到容器中，registrator 会监听容器的启动和停止，然后将容器注册到后端的数据存储，或者对其进行注销操作。现在有几种后端存储方式可供使用，比如 `etcd`（https://github.com/coreos/etcd）、Consul（https://www.consul.io）和 SkyDNS 2，registrator 将来也会支持更多的存储后端。尽管 `etcd` 是捆绑到 CoreOS 发布版本中的（参见范例 6.3），这些服务注册并不特定于 Docker。

要想使用 registrator，首先需要设置一个用于服务注册的存储后端。由于这些软件都以静态二进制文件的形式发布，你可以直接下载这些文件，然后以前台运行的方式进行测试。比如，要想使用 `etcd`，可以执行下面的命令。

```
$ curl -L  https://github.com/coreos/etcd/releases/download/v0.4.6/\
           etcd-v0.4.6-linux-amd64.tar.gz
       -o etcd-v0.4.6-linux-amd64.tar.gz
$ tar xzvf etcd-v0.4.6-linux-amd64.tar.gz
$ cd etcd-v0.4.6-linux-amd64
$ sudo ./etcd
2015/03/26 14:02:21 no data-dir provided, using default data-dir ./default.etcd
2015/03/26 14:02:21 etcd: listening for peers on http://localhost:2380
2015/03/26 14:02:21 etcd: listening for peers on http://localhost:7001
2015/03/26 14:02:21 etcd: listening for client requests on http://localhost:2379
2015/03/26 14:02:21 etcd: listening for client requests on http://localhost:4001
...
```

保持 `etcd` 处于运行状态。在其他的终端会话中，在 `etcd` 键值存储中创建一个文件夹（比如下面操作中的 cookbook）。当服务被发现之后，就会保存到这个文件夹之下。

```
$ cd etcd-v0.4.6-linux-amd64
$ ./etcdctl mkdir cookbook
$ ./etcdctl ls
/cookbook
```

然后从 Docker Hub 下载 registrator 镜像并启动它，如下所示。

```
$ docker pull gliderlabs/registrator
$ docker run -d -v /var/run/docker.sock:/tmp/docker.sock
               -h 192.168.33.10
               gliderlabs/registrator
               -ip 192.168.33.10
               etcd://192.168.33.10:4001/cookbook
```

启动容器时，你将注册服务的存储后端作为参数传给了 gliderlabs/registrator 镜像。别忘了你通过 `etcdctl` 命令创建的文件夹名。IP 地址和文件夹路径一起构成了后端存储服务端点的 URI。

将 `192.168.33.10` 替换为你自己环境中的 IP 地址。在这个例子中，我让所有的组件都在同一台主机上运行。但是，你可能更希望在与 registrator 所在的 Docker 主机集群独立的计算机上运行 `etcd`。

现在就可以启动容器了，将容器的端口暴露到宿主机上，然后就可以在 `etcd` 键值存储中看到容器的注册信息了，如下所示。

```
$ docker run -d -p 80:80 nginx
$ ./etcdctl ls /cookbook
/cookbook/nginx-80
$ ./etcdctl ls /cookbook/nginx-80
/cookbook/nginx-80/192.168.33.10:pensive_franklin:80
$ ./etcdctl get /cookbook/nginx-80/192.168.33.10:pensive_franklin:80
192.168.33.10:80
```

如果检查一下 registrator 容器的日志，你将会看到这个容器正在监听所有的 Docker 事件，然后对暴露到宿主机的端口进行注册，如下所示。

```
$ docker logs <CONTAINER_ID>
2015/03/26 ... registrator: Forcing host IP to 192.168.33.10
2015/03/26 ... registrator: Using etcd registry backend at \
                                  etcd://192.168.33.10:4001/cookbook
2015/03/26 ... registrator: ignored: 6f8043d9973f no published ports
2015/03/26 ... registrator: Listening for Docker events...
2015/03/26 ... registrator: ignored 8c033ca03a82 port 443 not published on host
2015/03/26 ... registrator: added: 8c033ca03a82 192.168.33.10:pensive_franklin:80
```

在上面的日志中，`6f8043d9973f` 是 registrator 容器的 ID，`8c033ca03a82` 是刚启动的 Nginx 容器的 ID。

7.13.3 讨论

存储在 `etcd` 中的键的命名规则取决于 registrator 创建的 `Service` 对象，并传递给注册后端。根据 GitHub 上的代码（https://github.com/gliderlabs/registrator），`Service` 结构定义如下所示。

```
type Service struct {
    ID    string               // <hostname>:<container-name>:<internal-port>
                               //[:udp if udp]
    Name  string               // <basename(container-image)>
                               //[-<internal-port> if >1 published ports]
    Port  int                  // <host-port>
    IP    string               // <host-ip> || <resolve(hostname)> if 0.0.0.0
    Tags  []string             // empty, or includes 'udp' if udp
    Attrs map[string]string    // any remaining service metadata from environment
}
```

服务的键规则定义如下所示。

```
<registry-uri-path>/<service-name>/<service-id>
```

在这个例子中，键的命名如下所示（请参考 `Service` 对象的 ID 定义）。

```
cookbook/nginx-80/192.168.33.10:pensive_franklin:80
```

然后这个键对应的值会被设置为 `<ip>:<port>`，在这个例子里就是 `192.168.33.10:80`（请参考 `Service` 对象中 IP 和 Port 的定义）。

如果你不想使用 `etcd`，可以切换注册后端存储，比如使用 `consul`（http://consul.io）。使用 Docker Hub 上的 progrium/consul 镜像，你可以在单台主机上轻松地尝试使用 `consul` 作为注册后端。在一个终端会话中拉取该镜像并启动 `consul` 代理（在这个例子中，容器没有以后台方式运行），如下所示。

```
$ docker pull progrium/consul
$ docker run -p 8400:8400 -p 8500:8500 -p 8600:53/udp
             -h cookbook progrium/consul -server
             -bootstrap -ui-dir /ui
```

在另一个终端会话中，启动 registrator 容器，但是将注册后端的 URI 更改为 consul://192.168.33.10:8500/foobar，如下所示。

```
$ docker run -d -v /var/run/docker.sock:/tmp/docker.sock
             -h 192.168.33.10 gliderlabs/registrator
             -ip 192.168.33.10 consul://192.168.33.10:8500/foobar
```

现在你可以启动一个 Nginx 容器，如下所示。

```
$ docker run -d -p 80:80 nginx
```

这时如果你通过 http://192.168.33.10:8500/ui 检查一下 Consul UI，你将会看到已经创建了一个 foobar 文件夹，里面有一些键，包括 Consul 容器自身用的键和 Nginx 容器的键，如图 7-9 所示。

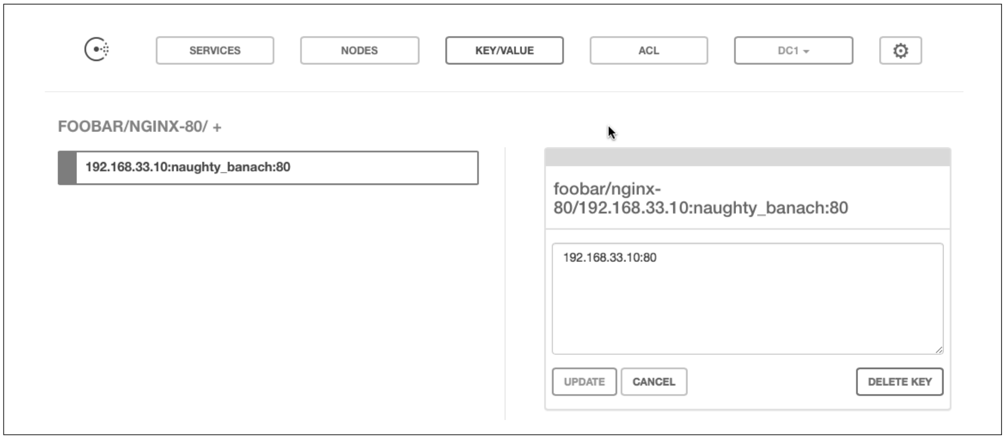


图 7-9：Consul UI

在掌握了 Docker 服务注册之后，你就可以开始思考如何动态地对其他服务进行重新配置了（参见范例 10.3）。

7.13.4　参考

- registrator GitHub 仓库（https://github.com/gliderlabs/registrator）
- Jeff Lindsay 的原始博客（http://progrium.com/blog/2014/09/10/automatic-docker-service-announcement-with-registrator/）

第 8 章

云计算中的Docker

8.0 简介

随着公有云和私有云的出现，企业已经把越来越多的工作负载转移到云中。IT 基础设施的一些重要部分都已经被部署到了公有云上，比如 AWS（http://aws.amazon.com）、GCE（https://cloud.google.com）和 Microsoft Azure（http://azure.microsoft.com/en-us/）。此外，企业也部署了私有云提供自服务的基础设施，以满足 IT 需求。

尽管 Docker 像其他软件一样，运行在裸机上，运行在位于公有云或者私有云的 Docker 主机（即虚拟机）上，但是对在这些 Docker 主机上运行的容器进行编排也成为了 IT 基础设施新需求中重要的一部分。图 8-1 描述了一个简单的构成，你可以使用本地 Docker 客户端访问位于云中的远程 Docker 主机。

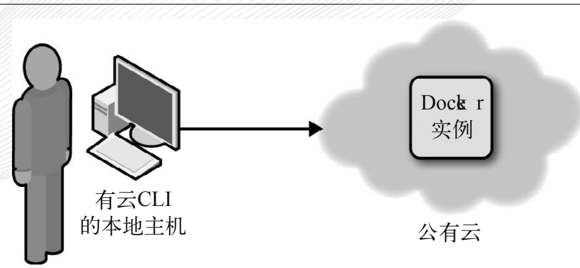


图 8-1：在云中使用 Docker

本章将会涉及最大的三家公有云（即 AWS、GCE 和 Azure）及其所提供的 Docker 服务。如果你从来没有使用过公有云，现在正是好时机，范例 8.1 将会帮你学习如何开始使用公

有云。然后，在范例 8.2、范例 8.3 和范例 8.4 中，你将会学到如何使用这些云服务提供的 CLI（命令行工具）来启动云主机实例并在其中安装 Docker。为了免去安装 CLI 的麻烦，我们会在范例 8.7 中为你介绍一个小窍门，就是让所有的云客户端在一个容器中运行。

虽然 Docker Machine（参见范例 1.9）最终会取代这些云计算服务的 CLI，但是学习如何使用这些 CLI 启动云主机实例有助于我们使用其他与 Docker 相关的云服务。尽管如此，在范例 8.5 中，我们还是会介绍如何使用 `docker-machine` 在 AWS EC2 中启动 Docker 主机，在范例 8.6 中会介绍如何在 Azure 中做同样的事。

然后我们会介绍一些在 GCE 和 CE2 上与 Docker 相关的服务。首先，在 GCE 中，我们会看一下 Google registry，一个托管的 Docker registry，你可以通过你的 Google 账号来使用这个 registry。它的工作原理就像 Docker Hub，但是同时利用了 Google 授权系统的优势，你可以为团队成员授予访问镜像的权限，如果你愿意，也可以将镜像公开。范例 8.9 会对 Google 容器虚拟机进行介绍。这里会对在单台主机上使用 Kubernetes 的一些概念进行简短的介绍。Google 容器引擎（即 GKE）是受托管的 Kubernetes 服务，在范例 8.10 中我们会对其进行介绍。如果你已经有了 Google 账号，那么 GKE 是体验 Kubernetes 最快的方式了。

本章最后，我们将看一下 AWS 提供的两个可以让你运行容器的服务。首先我们会在范例 8.11 中讲一下 Amazon Container Service（即 ECS，https://aws.amazon.com/ecs/）。在范例 8.12 中我们会讲解如何创建 ECS 集群，在范例 8.13 中讲解如何通过定义任务来运行容器。最后在范例 8.14 中，我们会演练一下如何使用 AWS Beanstalk 部署你的容器。

在本章中我们会向你展示如何使用公有云创建 Docker 主机，也会介绍一些最近已经面向一般用户开放的基于容器的服务：AWS 容器服务和 Google 容器引擎。这两种服务都标志着公有云提供商的一种新趋势，就是接受将 Docker 看作一种打包、部署和管理分布式应用的新方式。我们可以期待会有更多类似这样的服务出现，这些服务一般也会对 Docker 和容器的功能进行扩展。

> AWS、GCE 和 Azure 是世界公认的三大公有云提供商。但是，只要某公有云能安装 Docker 支持的 Linux 发行版（比如 Ubuntu、CentOS 和 CoreOS），Docker 就可以在该公有云上安装。

8.1　在公有云中运行Docker

8.1.1　问题

你需要访问公有云，并在云主机实例中运行 Docker。如果你从来没有使用过公有云服务，那么作为入门，你需要快速体验一下。

8.1.2 解决方案

如果还没有访问过公有云服务，你需要从下面这些云计算提供商中选择一个并创建账号。

- 如果是 GCE，你可以选择从免费试用（https://cloud.google.com/）开始。你需要一个 Google 账号，然后登录 GCE 管理控制台（https://cloud.google.com/console）。
- 如果是 Azure，你可以从免费试用（http://azure.microsoft.com/en-us/pricing/free-trial/）开始。
- 如果是 AWS，你可以从免费计划（free tier，http://aws.amazon.com/free/）开始。创建完账号之后，可以登录到 AWS 管理控制台（https://aws.amazon.com/console）。

登录到你选择的云计算服务提供商的 Web 控制台之后，找到启动云主机实例的向导页面。请确认你能启动一个可以通过 `ssh` 连接的云主机实例。图 8-2 显示的是 AWS 的管理控制台，图 8-3 显示的是 GCE 的管理控制台，图 8-4 显示的是 Azure 的管理控制台。

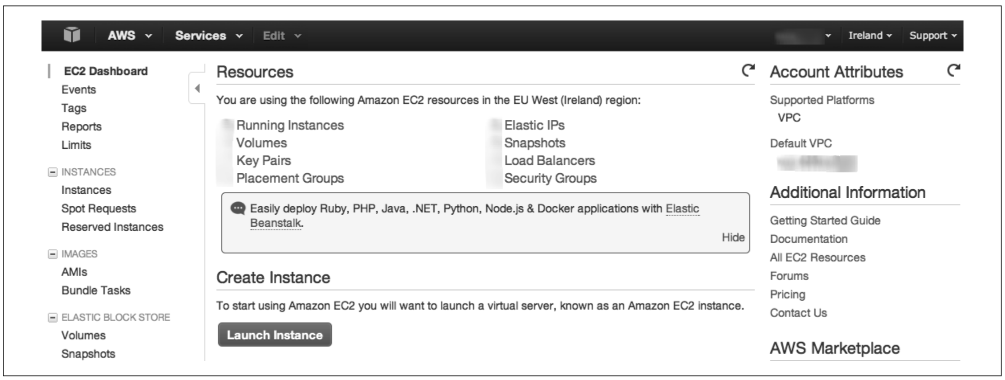


图 8-2：AWS 管理控制台

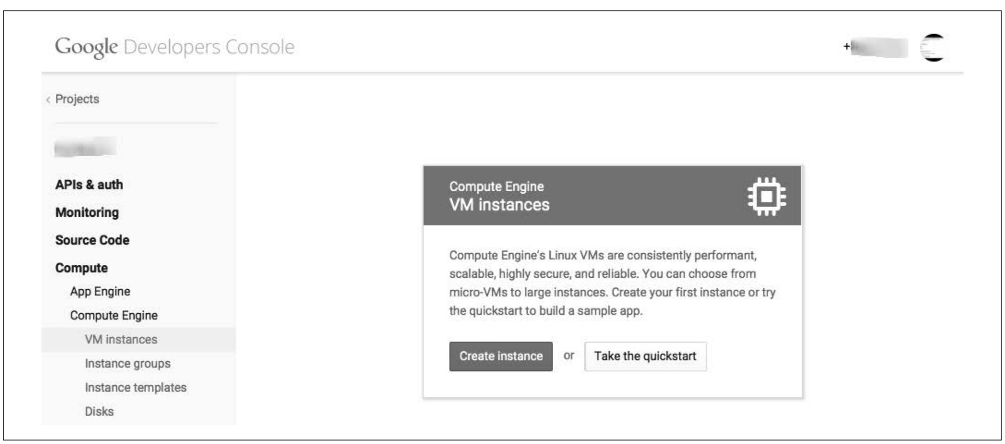


图 8-3：GCE 管理控制台

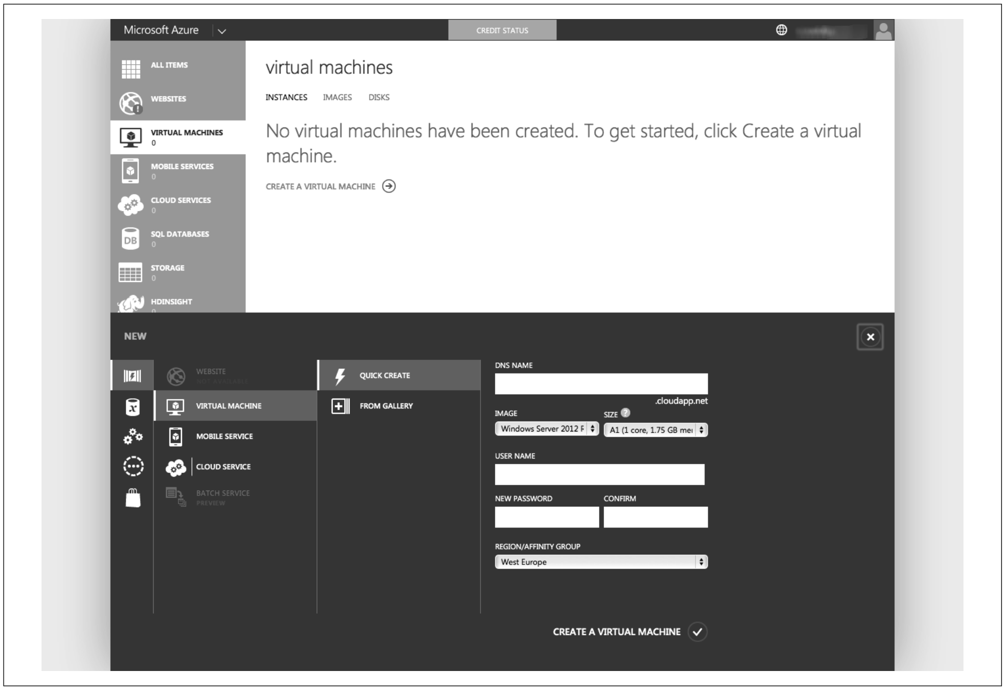
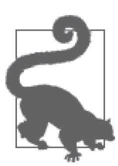


图 8-4：Azure 管理控制台

8.1.3 讨论



如果你不熟悉这些云计算服务提供商，并且没有完成上面的设置，那么将无法继续本章后面的范例学习。然而，提供这些云计算服务提供商完整的、详细到每一步的使用说明已经超出了本书的范围。



这些指令与使用 Docker 无关。当你在其中一个云计算服务中创建了账号之后，就可以访问该该云计算提供商的所有云服务了。

在 AWS 中，本章的范例将会使用 Elastic Compute Cloud（即 EC2，http://aws.amazon.com/documentation/ec2/）服务。要想启动云主机实例，你需要熟悉下面四条基本原则。

- 使用 AWS 命令行接口（command-line interface，CLI）需要的 API 密钥（http://docs.aws.amazon.com/cli/latest/userguide/cli-chap-getting-set-up.html）
- 用于通过 `ssh` 连接到云主机实例的 SSH 密钥对（http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-key-pairs.html）

- 用于控制 EC2 实例通信的安全组（Security group，http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-network-security.html）
- 在云主机实例启动时对实例进行配置的实例用户数据（http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/user-data.html）

在 GCE 中，你将要使用 Google Compute Engine 服务（https://cloud.google.com/compute/docs/）。上面的 AWS 原则也同样适用于 GCE。

- GCE 身份验证（https://cloud.google.com/compute/docs/authentication）。本章将会使用 `gcloud` 命令行工具，这个工具使用了 OAuth2 身份验证。GCE 也支持其他身份验证和授权机制。
- 使用 SSH 密钥（https://cloud.google.com/compute/docs/instances#sshkeys）连接到云主机实例
- 云主机实例的防火墙（https://cloud.google.com/compute/docs/networking#addingafirewall）
- 云主机实例的元数据（https://cloud.google.com/compute/docs/metadata#updatinginstancemetadata）。

8.1.4 参考

- *Programming Amazon Web Services*（http://shop.oreilly.com/product/9780596515812.do）
- AWS 入门指南（http://aws.amazon.com/documentation/gettingstarted/）
- *Automating Microsoft Azure Infrastructure Services*（http://shop.oreilly.com/product/0636920032380.do）
- GC 入门指南（https://cloud.google.com/compute/docs/signup）

8.2 在AWS EC2上启动Docker主机

8.2.1 问题

你希望启动一台 AWS EC2，并将它作为 Docker 主机使用。

8.2.2 解决方案

尽管你可以通过 Web 控制台启动云主机实例并在实例中安装 Docker，但是这里你将要使用 AWS CLI。首先，正如在范例 8.1 中所提到的那样，你需要获得一组 API 访问密钥。在 Web 控制台中，单击页面右上角你的账户名称，然后进入 Security Credentials 页面，如图 8-5 所示。在这里，你可以创建新的访问密钥。访问密钥和对应的密钥只会显示一次，所以你需要安全地保存这些信息。

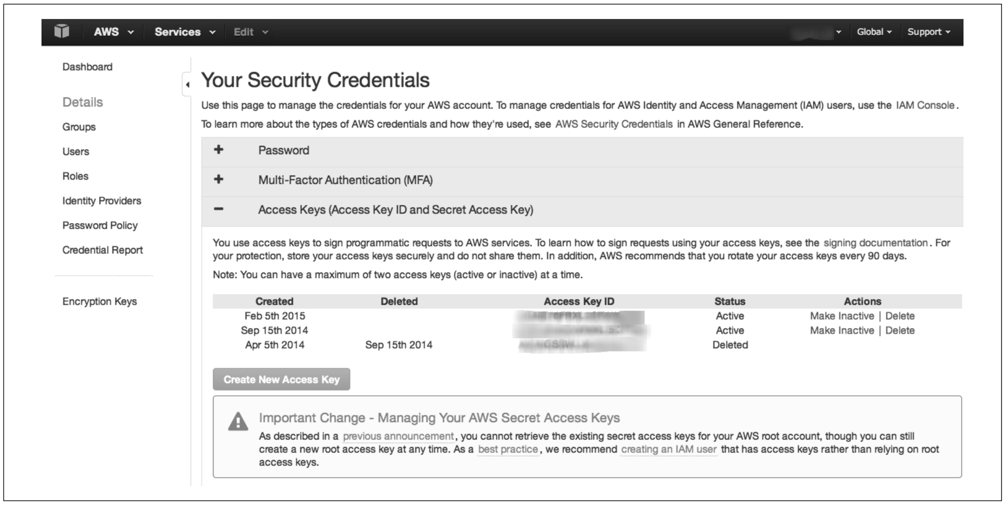


图 8-5：AWS Security Credentials 页面

然后你就可以安装 AWS CLI，并使用新创建的访问密钥对客户端进行配置。选择一个 AWS 可用区（http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-regions-availability-zones.html），你的云主机实例默认会在这个可用区创建。

AWS 的 CLI `aws` 是一个 Python 包，你可以通过 Python Package Index（`pip`）安装。比如，在 Ubuntu 上可以像下面这样安装。

```
$ sudo apt-get -y install python-pip
$ sudo pip install awscli
$ aws configure
AWS Access Key ID [****************n-mg]: AKIAIEFDGHQRTW3MNQ
AWS Secret Access Key [****************UjEg]: b4pWYhMUosg976arg9869Qd+Yg1qo22wC
Default region name [eu-east-1]: eu-west-1
Default output format [table]:
$ aws --version
aws-cli/1.7.4 Python/2.7.6 Linux/3.13.0-32-generic
```

要想通过 `ssh` 访问你的云主机实例，需要在 EC2 中配置一个 SSH 密钥对。可以通过 CLI 创建一个密钥对，将创建的私钥复制到 ~/.ssh 文件夹下的某个文件中，然后设置其权限为只有你能读和写。你需要确认密钥是否已创建，这可以通过 CLI 或者 Web 控制台来确认，如下所示。

```
$ aws ec2 create-key-pair --key-name cookbook
$ vi ~/.ssh/id_rsa_cookbook
$ chmod 600 ~/.ssh/id_rsa_cookbook
$ aws ec2 describe-key-pairs
-------------------------------------------------------------------------------
|                             DescribeKeyPairs                                |
+-----------------------------------------------------------------------------+
||                                KeyPairs                                   ||
```

```
|+--------------------------------------------------------------+-----------+|
||                        KeyFingerprint                        |  KeyName  ||
|+--------------------------------------------------------------+-----------+|
||  69:aa:64:4b:72:50:ee:15:9a:da:71:4e:44:cd:db:c0:a1:72:38:36  |  cookbook ||
|+--------------------------------------------------------------+-----------+|
```

现在你已经准备好在 EC2 上启动云主机实例了。AWS 提供的标准 Linux 镜像已经包含了 Docker 仓库。因此，当你使用 Amazon Linux AMI 启动 EC2 实例之后，离运行 Docker 就只有一步之遥（`sudo yum install docker`）了。

> 使用半虚拟化（paravirtualized，PV）的 Amazon Linux AMI，这样你就可以选择 `t1.micro` 实例类型了。另外，默认的安全组允许你通过 `ssh` 连接到云主机实例，所以如果你只需要通过 `ssh` 连接到主机实例，则不必在安全组中创建额外的规则。

```
$ aws ec2 run-instances --image-id ami-7b3db00c
                        --count 1
                        --instance-type t1.micro
                        --key-name cookbook
$ aws ec2 describe-instances
$ ssh -i ~/.ssh/id_rsa_cookbook ec2-user@54.194.31.39
The authenticity of host '54.194.31.39 (54.194.31.39)' can't be established.
RSA key fingerprint is 9b:10:32:10:ac:46:62:b4:7a:a5:94:7d:4b:2a:9f:61.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '54.194.31.39' (RSA) to the list of known hosts.

       __|  __|_  )
       _|  (     /   Amazon Linux AMI
      ___|\___|___|

https://aws.amazon.com/amazon-linux-ami/2014.09-release-notes/
[ec2-user@ip-172-31-8-174 ~]$
```

安装 Docker 软件包，启动 Docker 守护进程，并确认 Docker CLI 是否能正常工作。

```
[ec2-user@ip-172-31-8-174 ~]$ sudo yum update
[ec2-user@ip-172-31-8-174 ~]$ sudo yum install docker
[ec2-user@ip-172-31-8-174 ~]$ sudo service docker start
[ec2-user@ip-172-31-8-174 ~]$ sudo docker ps
CONTAINER ID    IMAGE       COMMAND     CREATED     STATUS      PORTS     NAMES
```

别忘了最后终止云主机实例，否则你可能会需要为此付费，如下所示。

```
$ aws ec2 terminate-instances --instance-ids <instance id>
```

8.2.3　讨论

在本范例中，我们花了一些时间创建了 API 访问密钥并安装了 CLI。值得高兴的是，不难发现，在 AWS 中创建 Docker 主机非常简单。现在使用标准的 AMI 安装 Docker 只需要两条命令。

Amazon Linux AMI 也包含 `cloud-init`（https://cloudinit.readthedocs.org/en/latest/），这已经变成了在系统初始化时对云主机实例进行配置的标准。它允许你在实例创建的时候传递用户数据。`cloud-init` 解析用户数据的内容并执行其中的命令。使用 AWS CLI，你可以传递一些用户数据来自动安装 Docker。不过它有个小缺点，就是需要 base64 编码。

基于前面的两条命令创建一个简单的 bash 脚本，如下所示。

```
#!/bin/bash
yum -y install docker
service docker start
```

对这个脚本进行 base64 编码并传递给实例创建命令，如下所示。

```
$ udata="$(cat docker.sh | base64 )"
$ aws ec2 run-instances --image-id ami-7b3db00c \
                        --count 1 \
                        --instance-type t1.micro \
                        --key-name cookbook \
                        --user-data $udata
$ ssh -i ~/.ssh/id_rsa_cookbook ec2-user@<public IP of the created instance>
$ sudo docker ps
CONTAINER ID    IMAGE       COMMAND     CREATED     STATUS      PORTS     NAMES
```

在 Docker 守护进程运行时，如果你想远程访问它，需要进行 TLS 设置（参见范例 4.9），并在你的安全组中打开 2376 端口。

使用 CLI 并不是 Docker 要求的。CLI 允许你使用完整的 AWS API 集合。但是使用 CLI 来启动云主机实例和在实例中安装 Docker 大大简化了 Docker 主机的配置工作。

8.2.4　参考

- 安装 AWS CLI（http://docs.aws.amazon.com/cli/latest/userguide/installing.html）
- 配置 AWS CLI（http://docs.aws.amazon.com/cli/latest/userguide/cli-chap-getting-started.html）
- 使用 AWS CLI 启动云主机实例（http://docs.aws.amazon.com/cli/latest/userguide/cli-ec2-launch.html）

8.3　在Google GCE上启动Docker主机

8.3.1　问题

你希望在 Google GCE 上启动一个虚拟机实例并将它作为 Docker 主机。

8.3.2 解决方案

安装 gcloud CLI（https://cloud.google.com/sdk/gcloud/），在安装过程中你需要回答几个问题，然后登录到 Google 云服务。如果 CLI 能打开浏览器，那么你会被重定向到一个网页，该页面会要求你登录并接受使用条款。如果你的终端不能启动浏览器，那么你会看到一个 URL，你需要自己在浏览器中打开这个页面。这将会给你一个需要在命令行提示符中输入的访问令牌，如下所示。

```
$ curl https://sdk.cloud.google.com | bash
$ gcloud auth login
Your browser has been opened to visit:
    https://accounts.google.com/o/oauth2/auth?redirect_uri=...
...
$ gcloud compute zones list
NAME           REGION       STATUS    NEXT_MAINTENANCE TURNDOWN_DATE
asia-east1-c   asia-east1   UP
asia-east1-a   asia-east1   UP
asia-east1-b   asia-east1   UP
europe-west1-b europe-west1 UP
europe-west1-c europe-west1 UP
us-central1-f  us-central1  UP
us-central1-b  us-central1  UP
us-central1-a  us-central1  UP
```

如果你还没有设置项目，需要先在 Web 控制台（https://cloud.google.com/storage/docs/projects）中设置一个。项目用于管理团队成员并为每个成员分配指定的权限。它大致相当于 Amazon 的身份和访问管理（Identity and Access Management，IAM）服务。

启动实例之前，为 region 和 zone（https://cloud.google.com/compute/docs/zones）设置一个你优先使用的默认值会比较方便（即使你想在云中部署一个将涉及多个 region 和 zone 的强大系统）。你可以使用 gcloud config set 来设置默认值，如下所示。

```
$ gcloud config set compute/region europe-west1
$ gcloud config set compute/zone europe-west1-c
$ gcloud config list --all
```

要想启动一个云主机实例，你需要一个镜像名（https://cloud.google.com/sdk/gcloud/reference/compute/instances/create）和实例类型（https://cloud.google.com/compute/docs/machine-types）。然后将剩下的工作交给 gcloud 工具，如下所示。

```
$ gcloud compute instances create cookbook \
                 --machine-type n1-standard-1 \
                 --image ubuntu-14-04 \
                 --metadata startup-script=\
                                "sudo wget -qO- https://get.docker.com/ | sh"
...
$ gcloud compute ssh cookbook
sebastiengoasguen@cookbook:~$ sudo docker ps
CONTAINER ID    IMAGE    COMMAND     CREATED     STATUS    PORTS    NAMES
...
$ gcloud compute instances delete cookbook
```

在这个例子中，我们创建了一个 Ubuntu 14.04 主机实例，主机类型为 `n1-standard-1`，并指定了作为启动脚本的元数据。指定的 bash 命令会从标准的 Ubuntu 仓库安装 docker.io 包。上面的例子创建了一个运行中的云主机实例，Docker 在这个云主机中运行。GCE 的元数据有点类似 AWS EC2 中的用户数据，在云主机实例中由 `cloud-init` 进行处理。

8.3.3 讨论

如果你列出一个 zone 中的可用镜像，会发现一些与 Docker 相关的有趣内容，如下所示。

```
$ gcloud compute images list
NAME                                PROJECT           ALIAS            ... STATUS
...
centos-7-v20150710                  centos-cloud      centos-7             READY
...
coreos-alpha-774-0-0-v20150814      coreos-cloud                           READY
...
container-vm-v20150806              google-containers container-vm         READY
...
ubuntu-1404-trusty-v20150805        ubuntu-os-cloud   ubuntu-14-04         READY
...
windows-server-2012-r2-dc-v20150813 windows-cloud     windows-2012-r2      READY
...
```

实际上，GCE 提供了 CoreOS（http://coreos.com）镜像以及容器 VM（https://cloud.google.com/compute/docs/containers/container_vms）。我们在第 6 章中已经讨论过 CoreOS 了。容器 VM 是基于 Debian 7 的实例，它包含 Docker 守护进程和 Kubernetes（http://kubernetes.io）的 kubelet。我们在第 5 章中已经讨论过了 Kubernetes，范例 8.9 则会对容器 VM 进行更详细的讲解。

如果你想启动 CoreOS 实例，可以使用镜像别名。安装 Docker 时你不需要指定任何元数据，如下所示。

```
$ gcloud compute instances create cookbook --machine-type n1-standard-1 \
                                           --image coreos
$ gcloud compute ssh cookbook
...
CoreOS (stable)
sebastiengoasguen@cookbook ~ $ docker ps
CONTAINER ID    IMAGE    COMMAND    CREATED    STATUS    PORTS    NAMES
```

使用 `gcloud` CLI 并不是 Docker 要求的。这个 CLI 让你可以访问完整的 GCE API 集合。但是使用 CLI 来启动云主机实例和在实例中安装 Docker 大大简化了 Docker 主机的配置工作。

8.4　在Microsoft Azure上启动Docker主机

8.4.1　问题

你想在 Microsoft Azure 上启动一个 VM 实例并将它作为 Docker 主机。

8.4.2　解决方案

首先，你需要创建一个 Azure 账号（参见图 8-1）。如果你不想使用 Azure 门户（https://manage.windowsazure.com），也可以安装 Azure CLI。在一台崭新的 Ubuntu 14.04 计算机上，你可以像下面这样操作。

```
$ sudo apt-get update
$ sudo apt-get -y install nodejs-legacy
$ sudo apt-get -y install npm
$ sudo npm install -g azure-cli
$ azure -v
0.8.14
```

接着，你需要从 CLI 设置用于身份验证的账号信息。有几种方法（http://azure.microsoft.com/en-us/documentation/articles/xplat-cli/）可以用来设置账号信息。其中一种是从门户下载你的账号设置并将其导入到使用 CLI 的计算机中，如下所示。

```
$ azure account download
$ azure account import ~/Downloads/Free\
Trial-2-5-2015-credentials.publishsettings
$ azure account list
```

现在你就可以使用 Azure CLI 来启动 VM 实例了。选择一个位置和一个镜像，如下所示。

```
$ azure vm image list | grep Ubuntu
$ azure vm location list
info:    Executing command vm location list
+ Getting locations
data:    Name
data:    ----------------
data:    West Europe
data:    North Europe
data:    East US 2
data:    Central US
data:    South Central US
data:    West US
data:    East US
data:    Southeast Asia
data:    East Asia
data:    Japan West
info:    vm location list command OK
```

使用 `azure vm create` 命令创建一个可以用 `ssh` 访问并使用密码身份验证的云主机实例，如下所示。

```
$ azure vm create cookbook --ssh=22 \
                           --password #@$#%#@$ \
                           --userName cookbook \
                           --location "West Europe" \
                           b39f27a8b8c64d52b05eac6a62ebad85__Ubuntu-14_04_1-LTS \
                           -amd64-server-20150123-en-us-30GB
...
$ azure vm list
...
data:    Name      Status     Location     DNS Name               IP Address
data:    --------  ---------  -----------  ---------------------  -------------
data:    cookbook  ReadyRole  West Europe  cookbook.cloudapp.net  100.91.96.137
info:    vm list command OK
```

之后你就可以通过 ssh 连接到这个云主机实例，并像在范例 1.1 中所介绍的那样安装 Docker。

8.4.3 讨论

Azure CLI 还在活跃地开发中（https://msopentech.com/blog/2014/10/08/latest-updates-to-azure-cli/）。你也可以在 GitHub（https://github.com/Azure/azure-xplat-cli）上找到其源代码，并且 Docker Machine 也提供了 Azure 驱动程序（https://github.com/docker/machine#microsoft-azure）。

Azure CLI 也支持使用 azure vm docker create 命令来自动创建一台 Docker 主机，如下所示。

```
$ azure vm docker create goasguen -l "West Europe" \
                                  b39f27a8b8c64d52b05eac6a62ebad85__Ubuntu \
                                  -14_04_1-LTS-amd64-server-20150123-en-us \
                                  -30GB cookbook @#$%@#$%$
info:    Executing command vm docker create
warn:    --vm-size has not been specified. Defaulting to "Small".
info:    Found docker certificates.
...
info:    vm docker create command OK
$ azure vm list
info:    Executing command vm list
+ Getting virtual machines
data:    Name      Status     Location     DNS Name               IP Address
data:    --------  ---------  -----------  ---------------------  -------------
data:    goasguen  ReadyRole  West Europe  goasguen.cloudapp.net  100.112.4.136
```

通过上面的命令创建的云主机实例会自动启动 Docker 守护进程，并且你可以通过 Docker 客户端和 TLS 连接来访问该 Docker 守护进程，如下所示。

```
$ docker --tls -H tcp://goasguen.cloudapp.net:4243 ps
CONTAINER ID    IMAGE       COMMAND     CREATED     STATUS     PORTS    NAMES
$ docker --tls -H tcp://goasguen.cloudapp.net:4243 images
REPOSITORY  TAG   IMAGE ID    CREATED     VIRTUAL SIZE
```

使用该 CLI 并不是 Docker 要求的。这个 CLI 让你可以访问完整的 Azure API 集合。但是使用 CLI 来启动云主机实例和在实例中安装 Docker 大大简化了 Docker 主机的配置工作。

8.4.4　参考

- Azure 命令行接口（http://azure.microsoft.com/en-us/documentation/articles/xplat-cli/）
- 在 Azure 上启动 CoreOS 实例（https://coreos.com/docs/running-coreos/cloud-providers/azure/#via-the-cross-platform-cli）
- 在 Azure 上使用 Docker Machine（https://github.com/chanezon/azure-linux/blob/master/docker/machine.md）

8.5　在AWS上使用Docker Machine启动Docker主机

8.5.1　问题

你知道如何使 AWS CLI 在云中启动云主机实例以及如何安装 Docker（参见范例 8.2）。但是你想使用一种能与 Docker 用户体验集成到一起的流水线式过程。

8.5.2　解决方案

使用 Docker Machine（https://github.com/docker/machine）和它的 AWS EC2 驱动程序。

下载 Docker Machine 的二进制文件。设置相应的环境变量，这样 Docker Machine 就可以知道你的 AWS API 密钥和默认你想在哪个 VPC 中启动 Docker 主机。然后就可以使用 Docker Machine 启动云主机实例。Docker 会自动设置 TLS 连接信息，你可以通过 TLS 连接到位于 AWS 的远程 Docker 主机。在一台 64 位的 Linux 主机上，执行下面的命令。

```
$ sudo su
# curl -L https://github.com/docker/machine/releases/\
          download/v0.4.0/docker-machine_linux-amd64 > \
                  /usr/local/bin/docker-machine
# chmod +x docker-machine
# exit
$ export AWS_ACCESS_KEY_ID=<your AWS access key>
$ export AWS_SECRET_ACCESS_KEY_ID=<your AWS secret key>
$ export AWS_VPC_ID=<the VPC ID you want to start the instance in>
$ docker-machine create -d amazonec2 cookbook
INFO[0000] Launching instance...
INFO[0023] Waiting for SSH ...
...
INFO[0129] "cookbook" has been created and is now the active machine
INFO[0129] To connect: docker $(docker-machine config cookbook) ps
```

当这台云主机实例创建后，就可以使用本地 Docker 客户端来与这台云主机通信了。别忘了在使用完之后终止这个云主机实例。

```
$ eval "$(docker-machine env cookbook)" ps
CONTAINER ID    IMAGE       COMMAND     CREATED     STATUS      PORTS     NAMES
$ docker-machine ls
NAME       ACTIVE   DRIVER      STATE     URL
cookbook   *        amazonec2   Running   tcp://<IP_Docker_Machine_AWS>:2376
$ docker-machine kill cookbook
```

也可以使用 Docker Machine CLI 直接管理你的云主机，如下所示。

```
$ docker-machine -h
...
COMMANDS:
   active Get or set the active machine
   create Create a machine
   config Print the connection config for machine
   inspect  Inspect information about a machine
   ip   Get the IP address of a machine
   kill   Kill a machine
   ls   List machines
   restart  Restart a machine
   rm   Remove a machine
   env    Display the commands to set up the environment for
          the Docker client
   ssh    Log into or run a command on a machine with SSH
   start  Start a machine
   stop   Stop a machine
   upgrade  Upgrade a machine to the latest version of Docker
   url    Get the URL of a machine
   help, h  Shows a list of commands or help for one command
```

8.5.3　讨论

Docker Machine 包括一些适用于云计算服务提供商的驱动程序（https://github.com/docker/machine/tree/master/drivers）。我们已经展示了如何使用 Digital Ocean 驱动程序（参见范例 1.9），你将会在范例 8.6 中看到如何使用 Azure 驱动程序。

AWS 驱动程序需要你通过一些命令行参数来设置你的访问密钥、VPC、SSH 密钥对、云主机镜像和实例类型等。你可以像前面例子那样通过环境变量的方式来设置，或者直接在 `docker-machine` 命令行中指定，如下所示。

```
$ docker-machine create -h
...
OPTIONS:
   --amazonec2-access-key                 AWS Access Key [$AWS_ACCESS_KEY_ID]
   --amazonec2-ami                        AWS machine image [$AWS_AMI]
   --amazonec2-instance-type 't2.micro'  AWS instance type [$AWS_INSTANCE_TYPE]
   --amazonec2-region 'us-east-1'        AWS region [$AWS_DEFAULT_REGION]
```

```
--amazonec2-root-size '16'           AWS root disk size (in GB) ...
--amazonec2-secret-key               AWS Secret Key [$AWS_SECRET_ACCESS_KEY]
--amazonec2-security-group           AWS VPC security group ...
--amazonec2-session-token            AWS Session Token [$AWS_SESSION_TOKEN]
--amazonec2-subnet-id                AWS VPC subnet id [$AWS_SUBNET_ID]
--amazonec2-vpc-id                   AWS VPC id [$AWS_VPC_ID]
--amazonec2-zone 'a'                 AWS zone for instance ... [$AWS_ZONE]
```

最后 ,docker-machine 还会为你创建一个 SSH 密钥对和一个安全组。这个安全组会打开 2376 端口上的通信，这样就可以让 Docker 客户端通过 TLS 与 Docker 主机进行通信。图 8-6 显示了在 AWS 控制台中看到的安全组规则。

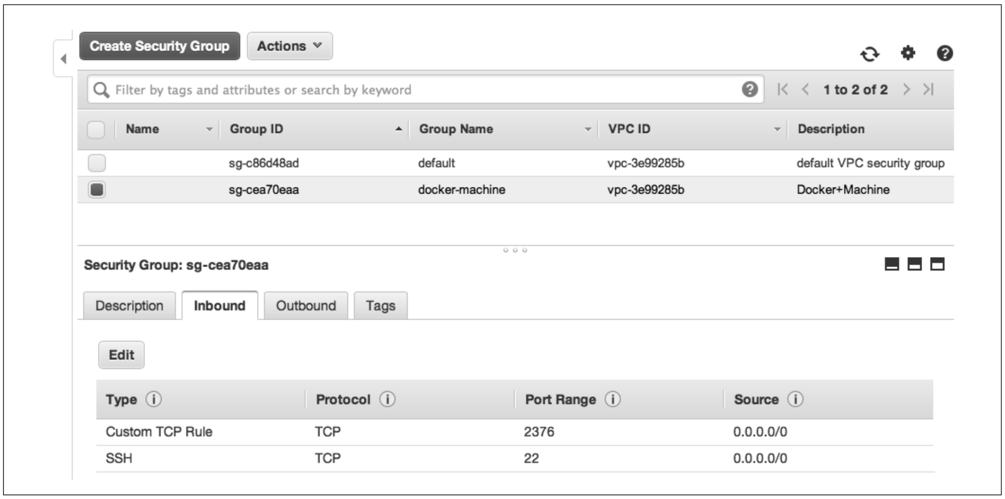


图 8-6：Docker Machine 创建的安全组规则

8.6 在Azure上使用Docker Machine启动Docker主机

8.6.1 问题

你知道如何通过 Azure CLI 在 Azure 上启动 Docker 主机，但是你想知道使用 Docker Machine 统一在多个公有云平台上启动 Docker 主机的方法。

8.6.2 解决方案

使用 Docker Machine 的 Azure 驱动程序。在图 1-7 中，你已经看到了如何通过 Docker Machine 在 DigitalOcean 上启动一台 Docker 主机。在 Microsoft Azure 上你也可以进行同样的操作。你需要一个 Azure 的合法订阅（http://azure.microsoft.com/en-us/pricing/free-trial/）。

你需要下载 docker-machine 可执行文件。访问 docker-machine 文档网站（https://docs.

docker.com/machine/)，然后选择与你本地计算机相应的架构可执行文件，比如在 OS X 上，如下所示。

```
$ wget https://github.com/docker/machine/releases/download/v0.4.0/ \
docker-machine_darwin-amd64
$ mv docker-machine_darwin-amd64 docker-machine
$ chmod +x docker-machine
$ ./docker-machine --version
docker-machine version 0.3.0
```

有了合法的 Azure 订阅，还需要创建一个 X.509 证书并通过 Azure 门户（https://manage.windowsazure.com）上传。可以通过下面的命令来创建证书。

```
$ openssl req -x509 -nodes -days 365 -newkey rsa:1024 \
              -keyout mycert.pem -out mycert.pem
$ openssl pkcs12 -export -out mycert.pfx -in mycert.pem -name "My Certificate"
$ openssl x509 -inform pem -in mycert.pem -outform der -out mycert.cer
```

将 mycert.cer 上传到服务并定义下面的环境变量。

```
$ export AZURE_SUBSCRIPTION_ID=<UID of your subscription>
$ export AZURE_SUBSCRIPTION_CERT=mycert.pem
```

然后就可以使用 `docker-machine` 并设置本地 Docker 客户端使用远程的 Docker 守护进程，如下所示。

```
$ ./docker-machine create -d azure goasguen-foobar
INFO[0002] Creating Azure machine...
INFO[0061] Waiting for SSH...
INFO[0360] "goasguen-foobar" has been created and is now the active machine.
INFO[0360] To point your Docker client at it, run this in your shell: \
$(docker-machine env goasguen-foobar)
$ ./docker-machine ls
NAME       ACTIVE   DRIVER   STATE     URL                                        SWARM
toto1111   *        azure    Running   tcp://goasguen-foobar.cloudapp.net:2376
$ $(docker-machine env goasguen-foobar)
$ docker ps
CONTAINER ID    IMAGE    COMMAND    CREATED    STATUS    PORTS    NAMES
```

在这个例子中，`goasguen-foobar` 是我为新创建的 Docker 主机设置的名称。这个名称需要全局唯一。像 foobar 和 test 这种名字很可能都已经被用了。

8.6.3 讨论

将你的本地 Docker 客户端设置为使用在 Azure 虚拟机中运行的远程 Docker 守护进程之后，就可以从你常用的 registry 拉取镜像并启动容器了。

比如，让我们启动一个 Nginx 容器，如下所示。

```
$ docker pull nginx
$ docker run -d -p 80:80 nginx
```

为了暴露位于 Azure 远程主机上的 80 端口，你需要为该 VM 添加一个端点。转到 Azure 门户，选择一个 VM（这里为 `goasguen-foobar`），为 HTTP 请求添加一个端点，如图 8-7 所示。当端点创建完成之后，你就可以通过 http://<unique_name>.cloudapp.net 访问 Nginx 了。

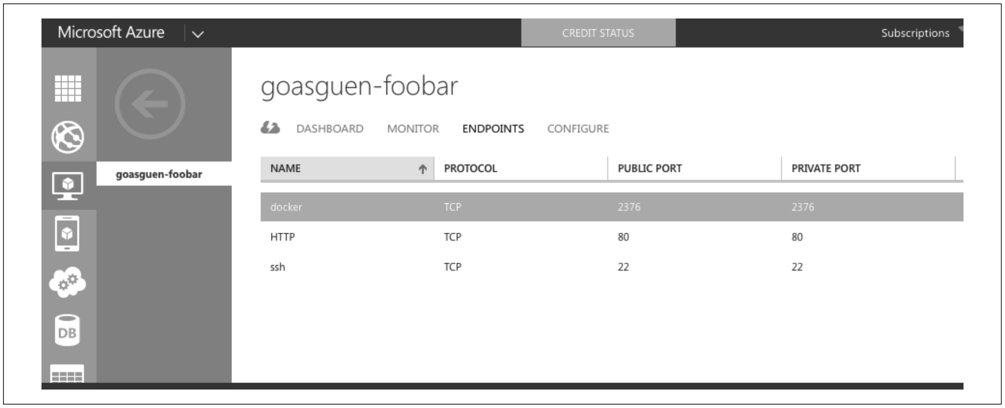


图 8-7：Azure 虚拟机端点

8.6.4　参考

- Docker Machine Azure 驱动程序文档（http://docs.docker.com/machine/#microsoft-azure）

8.7　在Docker容器中运行云服务提供商的CLI

8.7.1　问题

你想利用容器的优势，在容器内运行你选择的云服务提供商的 CLI。这给了你更多的可移植性，并免去了每次都从头安装 CLI 的麻烦。你只需要从 Docker Hub 下载一个容器的镜像就可以了。

8.7.2　解决方案

如果是 Google GCE CLI，你可以使用 Google 维护的公开镜像（https://registry.hub.docker.com/u/google/cloud-sdk/）。通过 `docker pull` 命令下载这个镜像，并通过临时的交互式容器执行 GCE 命令。

假设你在 OS X 计算机上使用 `boot2docker`，可以进行如下操作。

```
$ boot2docker up
$ $(boot2docker shellinit)
$ docker pull google/cloud-sdk
$ docker images | grep google
google/cloud-sdk    latest      a7e7bcdfdc16    10 days ago     1.263 GB
```

然后就可以登录到 GCE 并执行在范例 8.3 中介绍过的命令，唯一的不同是这里的 CLI 都在

容器中运行。运行 login 命令的容器设置了一个名称，在后续的 CLI 调用中，这个被命名的容器会被当作一个数据卷容器来使用（即 --volumes-from cloud-config）。这样你就可以在后续操作中使用存储在这个被命名容器中的授权令牌信息，如下所示。

```
$ docker run -t -i --name gcloud-config google/cloud-sdk gcloud auth login
Go to the following link in your browser:
...
$ docker run --rm \
             -ti \
             --volumes-from gcloud-config google/cloud-sdk \
             gcloud compute zones list
NAME           REGION       STATUS     NEXT_MAINTENANCE TURNDOWN_DATE
asia-east1-c   asia-east1   UP
asia-east1-a   asia-east1   UP
asia-east1-b   asia-east1   UP
europe-west1-b europe-west1 UP
europe-west1-c europe-west1 UP
us-central1-f  us-central1  UP
us-central1-b  us-central1  UP
us-central1-a  us-central1  UP
```

使用别名会更加方便，如下所示。

```
$ alias magic='docker run --rm \
                          -ti \
                          --volumes-from gcloud-config \
                          google/cloud-sdk gcloud'
$ magic compute zones list
NAME           REGION       STATUS     NEXT_MAINTENANCE TURNDOWN_DATE
asia-east1-c   asia-east1   UP
asia-east1-a   asia-east1   UP
asia-east1-b   asia-east1   UP
europe-west1-b europe-west1 UP
europe-west1-c europe-west1 UP
us-central1-f  us-central1  UP
us-central1-b  us-central1  UP
us-central1-a  us-central1  UP
```

8.7.3 讨论

你也可以在 AWS 上采用类似的过程。如果在 Docker Hub 上查找 awscli 镜像，你会发现有很多镜像可供选择。其中的一个 Dockerfile（https://registry.hub.docker.com/u/nathanleclaire/awscli/dockerfile/）显示了这个镜像是如何构建的，以及 CLI 是如何在镜像中安装的。使用这个 nathanleclaire/awscli 镜像，你会发现这个镜像并没有通过挂载卷的方式在容器和容器之间共享授权信息。因此，你需要在启动容器时通过环境变量的方式将 AWS 访问密钥传递给容器内的 CLI，如下所示。

```
$ docker pull nathanleclaire/awscli
$ docker run --rm \
             -ti \
             -e AWS_ACCESS_KEY_ID="AKIAIUCASDLGFIGDFGS" \
             -e AWS_SECRET_ACCESS_KEY="HwQdNnAIqrwy9797arghqQERfrgot" \
```

```
            nathanleclaire/awscli \
            --region eu-west-1 \
            --output=table \
            ec2 describe-key-pairs
-------------------------------------------------------------------------------
|                              DescribeKeyPairs                               |
+-----------------------------------------------------------------------------+
||                                 KeyPairs                                  ||
|+---------------------------------------------------------------+-----------+|
||                        KeyFingerprint                          |  KeyName  ||
|+---------------------------------------------------------------+-----------+|
||  69:aa:64:4b:72:50:ee:15:9a:da:71:4e:44:cd:db:c0:a1:72:38:36   |  cookbook ||
|+---------------------------------------------------------------+-----------+|
```

还需要注意的是，这个容器的 `entrypoint` 已经被设置为 `aws` 了，因此你不需要在运行时再输入 `aws`，只需要输入后面的参数即可。

> 你可以构建自己的 AWS CLI 镜像，这样就可以更方便地使用 API 密钥了。

8.7.4　参考

- 关于容器化 Google SDK 的官方文档（https://registry.hub.docker.com/u/google/cloud-sdk/）

8.8　使用Google Container registry存储Docker镜像

8.8.1　问题

你已经在你自己的基础设施中使用过私有 registry 了（参见范例 2.11），但是你想发挥托管服务的优势。特别是你想发挥新发布的 Google 容器 registry（https://cloud.google.com/tools/container-registry/）的优势。

> 除此之外，还有一些其他托管 registry 解决方案可供选择，比如 Docker Hub 企业版（https://www.docker.com/enterprise/hub/）和 Quay.io（https://quay.io）。本范例并不会提供其中任何一种方案与其他方案优劣的对比。

8.8.2　解决方案

如果还没有 GCE 账号，请回到范例 8.1 注册一个 Google 云计算平台的账号。然后下载 Google 云计算的 CLI，并创建一个项目（参见范例 8.3）。请务必在你的 Docker 主机上更新 `gcloud` CLI，让它能够加载预览组件。你将可以使用 `gcloud docker` 命令，这是一个

docker 客户端的包装工具，如下所示。

```
$ gcloud components update
$ gcloud docker help
Usage: docker [OPTIONS] COMMAND [arg...]

A self-sufficient runtime for linux containers.
...
```

这个例子会在 Google Cloud 上创建一个 cookbook 项目（https://cloud.google.com/storage/docs/projects），项目 ID 为 sylvan-plane-862。实际上，你创建的项目名和项目 ID 与例子中的可能会不一样。

在这个例子中，我们使用的 Docker 主机上有一个 busybox 镜像，我们将会把这个镜像上传到 Google Container Registry（GCR）上去。你需要遵循 GCR 的命名空间规则（即 `gcr.io/project_id/image_name`）为你要推送到 GCR 的镜像打上标签。然后可以通过 `gcloud docker push` 命令上传镜像，如下所示。

```
$ docker images | grep busybox
busybox    latest    a9eb17255234    8 months ago    2.433 MB
$ docker tag busybox gcr.io/sylvan_plane_862/busybox
$ gcloud docker push gcr.io/sylvan_plane_862/busybox
The push refers to a repository [gcr.io/sylvan_plane_862/busybox] (len: 1)
Sending image list
Pushing repository gcr.io/sylvan_plane_862/busybox (1 tags)
511136ea3c5a: Image successfully pushed
42eed7f1bf2a: Image successfully pushed
120e218dd395: Image successfully pushed
a9eb17255234: Image successfully pushed
Pushing tag for rev [a9eb17255234] on \
{https://gcr.io/v1/repositories/sylvan_plane_862/busybox/tags/latest}
```

> GCR 命名空间的命名规则是，如果项目 ID 中有破折号，你需要将破折号替换为下划线。

如果打开 Google 开发者控制台中的存储页面，你会看到一个新创建的 bucket，并且镜像中的层已经上传到服务器（参见图 8-8）。

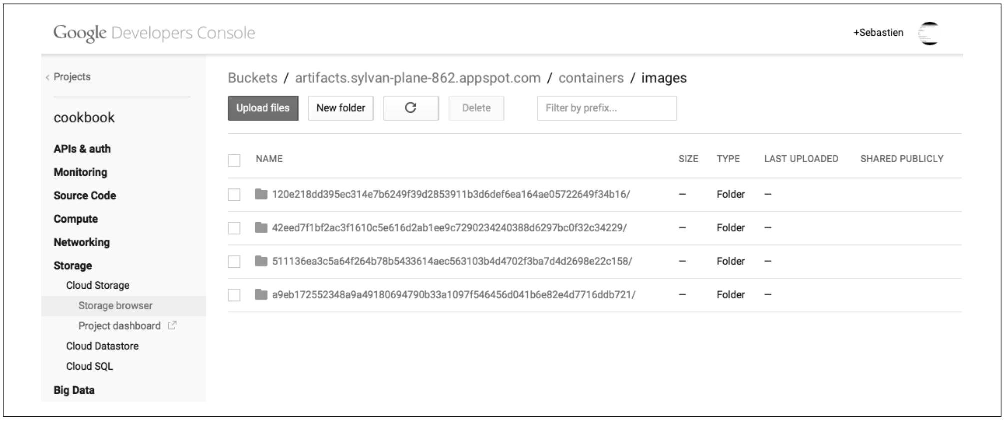


图 8-8：Google 容器 registry 镜像

8.8.3 讨论

如果你在镜像所在的项目下面启动新的 Google 云主机实例，那么将会自动获得拉取这个镜像的权限。如果你想让其他人也能拉取这个镜像，则需要将这些人添加到项目的成员中。你可以通过 `gcloud config set project <project_id>` 命令设置默认的项目，这样在后续的 `gcloud` 命令中，你就不需要再明确指定项目名参数了。

让我们在 GCE 中启动一个云主机实例，然后 `ssh` 到该实例，从 GCR 拉取 busybox 镜像，如下所示。

```
$ gcloud compute instances create cookbook-gce --image container-vm \
                                              --zone europe-west1-c \
                                              --machine-type f1-micro
$ gcloud compute ssh cookbook-gce
Updated [https://www.googleapis.com/compute/v1/projects/sylvan-plane-862].
...
$ sudo gcloud docker pull gcr.io/sylvan_plane_862/busybox
Pulling repository gcr.io/sylvan_plane_862/busybox
a9eb17255234: Download complete
511136ea3c5a: Download complete
42eed7f1bf2a: Download complete
120e218dd395: Download complete
Status: Downloaded newer image for gcr.io/sylvan_plane_862/busybox:latest
sebastiengoasguen@cookbook:~$ sudo docker images | grep busybox
gcr.io/sylvan_plane_862/busybox   latest    a9eb17255234    ...
```

为了能从 GCE 云主机实例推送镜像，你需要在启动时指定正确的 scope 参数：`--scopes https://www.googleapis.com/auth/devstorage.read_write`。

8.9 在GCE Google-Container实例中使用Docker

8.9.1 问题

你知道如何在 GCE 中启动云主机实例，并在云主机初始化时安装 Docker，但是你希望使用已经安装了 Docker 的云主机镜像。

8.9.2 解决方案

正如在范例 8.3 中提到的那样，GCE 提供了专为容器优化的镜像。

> 确保你通过 gcloud config set project <project_id> 命令将你的项目设置为项目 ID。

```
$ gcloud compute images list
NAME                              PROJECT           ALIAS        DEPRECATED STATUS
...
container-vm-v20141208            google-containers container-vm            READY
container-vm-v20150112            google-containers container-vm            READY
container-vm-v20150129            google-containers container-vm            READY
...
```

这些镜像（https://cloud.google.com/compute/docs/containers/container_vms）基于 Debian 7，预装了 Docker 守护进程和 Kubernetes（http://kubernetes.io）的 kubelet 服务。

> 第 5 章中对 Kubernetes 有更详细的讨论。

用户可以向在基于这些云主机镜像创建的云主机实例中运行的 kubelet 服务发送一个描述文件（即一个 pod：https://github.com/kubernetes/kubernetes/blob/master/docs/user-guide/pods.md），这个描述文件记述了一组需要在这个云主机实例中运行的容器。kubelet 将会启动这些容器并对这些容器进行管理。一个 pod 的描述文件是一个如下所示的 YAML 文件。

```
version: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
    - name: nginx
      image: nginx
      ports:
```

```
    - name: nginx
      hostPort: 80
      containerPort: 80
```

pod 描述文件中可以引用 GCR 中的镜像（参见范例 8.8），比如 `gcr.io/<your_project_name>/busybox`。

这个示例文件描述了一个基于 nginx 镜像的容器，以及一个要暴露的端口。你可以将这个描述文件作为参数传给 `gcloud` 创建云主机实例的命令。将上面的 YAML 文件保存为 nginx.yml，然后启动云主机实例，如下所示。

```
$ gcloud compute instances create cookbook-gce \
         --image container-vm \
         --metadata-from-file google-container-manifest=nginx.yml \
         --zone europe-west1-c \
         --machine-type f1-micro
```

在你的 Google GCE 管理控制台中，可以查看刚启动的云主机实例的详情（参见图 8-9）。在这个页面中你可以看到容器的描述文件，也可以设置是否允许 HTTP 通信。如果 API 的版本升级了，那么你看到的版本可能是 `v2` 而不是 `v1`。当 pod 描述文件定义的容器启动之后，打开浏览器访问这台云主机实例的 80 端口，你将会看到 Nginx 的欢迎页面。

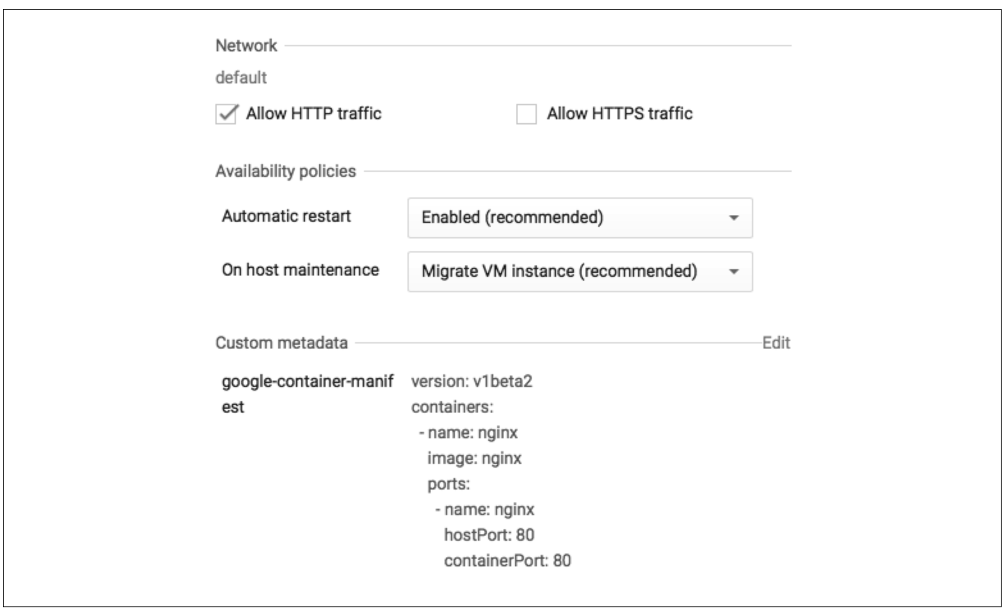


图 8-9：GCE 容器 VM 中的 pod 描述文件

8.9.3　讨论

如果通过 `ssh` 直接连接到云主机实例，你可以查看到运行中的容器列表。你将会看到一个用于监控的 google/cadvisor 容器，以及两个 kubernetes/pause:go 容器。后面的这两个容器则是监控用容器 cadvisor 和 pod 所暴露端口的中间代理。

```
$ gcloud compute ssh cookbook-gce
...
sebastiengoasguen@cookbook-gce:~$ sudo docker ps
CONTAINER ID        IMAGE                    COMMAND                ...
1f83bb1197c9        nginx:latest             "nginx -g 'daemon of   ...
b1e6fed3ee20        google/cadvisor:0.8.0    "/usr/bin/cadvisor"    ...
79e879c48e9e        kubernetes/pause:go      "/pause"               ...
0c1a51ab2f94        kubernetes/pause:go      "/pause"               ...
```

在第 9 章中将会介绍 cadvisor（https://github.com/google/cadvisor）。

8.10　通过GCE在云中使用Kubernetes

8.10.1　问题

你希望使用一组 Docker 主机并在这些 Docker 主机中管理容器。你喜欢 Kubernetes（https://kubernetes.io）容器编排引擎，但是想通过托管的云服务方式来使用 Kubernetes。

8.10.2　解决方案

使用 Google 容器引擎（https://cloud.google.com/container-engine/）服务。这个新服务允许你在需要的时候通过 Google API 来创建 Kubernetes 集群。一个 Kubernetes 集群由一个主节点和一组作为容器 VM 的计算节点组成，有点类似范例 8.9 中所介绍的那样。

> Google 容器引擎还处于 beta 阶段。Kubernetes 也在活跃开发中。可以预期其 API 可能会经常变动，因此在生产环境下使用 Kubernetes 你需要自己承担风险。关于 Kubernetes 的详细信息，可以参考第 5 章。

更新你的 `gcloud` SDK 以使用容器引擎功能。如果还没有安装 Google SDK，请参考范例 8.3 安装 Google SDK。

```
$ gcloud components update
```

通过 Google 容器引擎启动一个 Kubernetes 计算机只需要一条命令，如下所示。

```
$ gcloud container clusters create cook --num-nodes 1 --machine-type g1-small
Creating cluster cook...done.
Created [https://container.googleapis.com/v1/projects/sylvan-plane-862/zones/ \
us-central1-f/clusters/cook].
kubeconfig entry generated for cook.
NAME  ZONE           MASTER_VERSION  MASTER_IP      MACHINE_TYPE  STATUS
cook  us-central1-f  1.0.3           104.197.33.61  g1-small      RUNNING
```

你的集群 IP 地址、项目名和 zone 会与上面看到的有所不同。我们看到 Kubernetes 为你创建了配置文件 kubeconfig。这个文件保存在 ~/.kube/config 下，文件内容包括我们的容器集群的端点，以及使用 Kubernetes 集群所需要的授权信息。

你也可以通过 Google Cloud 的 Web 控制台（参见图 8-10）创建一个集群。

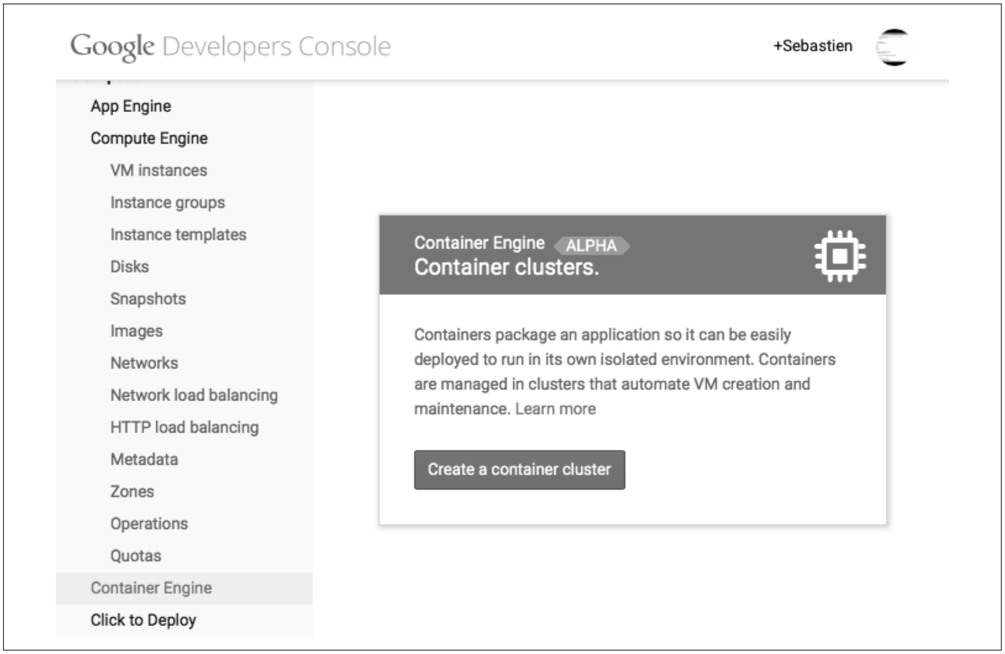


图 8-10：容器引擎向导

当集群开始运行之后，就可以向集群提交容器，也就是说，你可以通过与集群中的主节点交互，在集群中的节点上创建一组容器。容器组由 pod 来定义。这与范例 8.9 中介绍的是一样的概念。使用 `gcloud` CLI 可以方便地定义简单的 pod 并将 pod 提交到集群。接着你要使用 tutum/wordpress 镜像来启动一个容器，这个容器包括一个 MySQL 数据库。如果安装了 `gcloud` CLI，它同时也会安装 Kubernetes 客户端程序 `kubectl`。你可以确认一下 `kubectl` 是否在你的 `PATH` 之中。它将会使用创建 Kubernetes 集群时自动创建的配置文件。这样就可以从位于远程容器集群上的本地主机安全地启动容器，如下所示。

```
$ kubectl run wordpress --image=tutum/wordpress --port=80
$ kubectl get pods
NAME              READY     STATUS    RESTARTS   AGE
wordpress-0d58l   1/1       Running   0          1m
```

当这个容器被分配到集群中的某个节点后，你需要创建一个 Kubernetes 服务来将在容器中运行的应用程序暴露到外部。这也可以通过 `kubectl` 命令来完成，如下所示。

```
$ kubectl expose rc wordpress --create-external-load-balancer=true
NAME        LABELS          SELECTOR        IP(S)     PORT(S)
wordpress   run=wordpress   run=wordpress             80/TCP
```

命令 `expose` 用于创建一个 Kubernetes 服务（service、pod、replication controller 是 Kubernetes 的三个核心组件），该服务同时从负载平衡服务器获得了一个公网 IP 地址。完成上述操作后再次查看容器集群中的服务列表，你会看到 wordpress 服务。该服务有一个内部 IP 地址和一个公网 IP 地址，你可以在笔记本电脑上访问这个 WordPress 的界面，如下所示。

```
$ kubectl get services
NAME         LABELS                 SELECTOR        IP(S)            PORT(S)
wordpress   run=wordpress1         run=wordpress   10.95.252.182    80/TCP
                                                    104.154.82.185
```

之后就可以开始使用 WordPress 了。

8.10.3 讨论

你可以使用 `kubectl` CLI 来管理 Kubernetes 集群中的任何资源（即 pod、service、replication controller 和节点）。与下面这段 `kubectl` 使用帮助显示的内容一样，你可以创建、删除、查看和列出所有这些资源。

```
$ kubectl -h
kubectl controls the Kubernetes cluster manager.

Find more information at https://github.com/GoogleCloudPlatform/kubernetes.

Usage:
  kubectl [flags]
  kubectl [command]

Available Commands:
  get            Display one or many resources
  describe       Show details of a specific resource or group of resources
  create         Create a resource by filename or stdin
  replace        Replace a resource by filename or stdin.
  patch          Update field(s) of a resource by stdin.
  delete         Delete a resource by filename, stdin, resource and name, or ...
...
```

尽管通过上面的例子可以创建只包含一个容器的简单 pod，不过使用 `-f`，你还可以通过 JSON 或者 YAML 文件来指定更高级的 pod 定义文件，如下所示。

```
$ kubectl create -f /path/to/pod/pod.json
```

在范例 8.9 中你已经看到过 pod 的 YAML 定义文件了。这里再让我们通过 JSON 文件，使用新发布的 Kubernetes v1 API 来定义一个 pod。这个 pod 会运行 Nginx 服务，如下所示。

```
{
  "kind": "Pod",
  "apiVersion": "v1",
  "metadata": {
    "name": "nginx",
    "labels": {
      "app": "nginx"
    }
```

```
    },
    "spec": {
      "containers": [
        {
          "name": "nginx",
          "image": "nginx",
          "ports": [
            {
              "containerPort": 80,
              "protocol": "TCP"
            }
          ]
        }
      ]
    }
  }
```

启动 pod，并检查 pod 的状态。当 pod 开始运行，并且你的防火墙打开了集群中节点上的 80 端口时，就可以看到 Nginx 的欢迎页面。Kubernetes 的 GitHub 项目主页（https://github.com/GoogleCloudPlatform/kubernetes/tree/master/examples）提供了更多的例子。

```
$ kubectl create -f nginx.json
pods/nginx
$ kubectl get pods
NAME              READY     STATUS    RESTARTS   AGE
nginx             1/1       Running   0          20s
wordpress         1/1       Running   0          17m
```

最后，你需要做些清理工作，删除你的 pod，退出主节点，并删除集群，如下所示。

```
$ kubectl delete pods nginx
$ kubectl delete pods wordpress
$ gcloud container clusters delete cook
```

8.10.4　参考

- 集群操作（https://cloud.google.com/container-engine/docs/clusters/operations）
- pod 操作（https://cloud.google.com/container-engine/docs/pods/operations）
- service 操作（https://cloud.google.com/container-engine/docs/services/operations）
- replication controller 操作（https://cloud.google.com/container-engine/docs/services/operations）

8.11　配置使用EC2 Container Service

8.11.1　问题

你希望尝试一下 Amazon AWS 的最新 EC2 容器服务（EC2 Container Service，ECS）。

8.11.2　解决方案

ECS 是一个 AWS 公开发布的服务。设置 ECS 需要一些步骤，本范例将会总结一下其中

的主要步骤，但你应该阅读一下官方文档（http://docs.aws.amazon.com/AmazonECS/latest/developerguide/get-set-up-for-amazon-ecs.html）以获得更多信息。

(1) 如果你还没有 AWS（http://aws.amazon.com）账号，需要先注册一个。
(2) 登录到 AWS 管理控制台。如有必要，请参考一下范例 8.1 和范例 8.2。你需要在一个 VPC 内的安全组中启动 ECS 实例。如果还没有默认的 VPC 和安全组，就需要新建一个 VPC 和安全组。
(3) 到 IAM 管理控制台创建用于 ECS 的角色。如果你还不熟悉 IAM，那么这一步可能稍显高级，你可以按照 AWS ECS 的文档（http://docs.aws.amazon.com/AmazonECS/latest/developerguide/get-set-up-for-amazon-ecs.html）一步步进行操作。
(4) 为刚创建的角色创建一个内联策略（http://docs.aws.amazon.com/AmazonECS/latest/developerguide/get-set-up-for-amazon-ecs.html）。如果策略创建成功，那么当你单击 Show Policy 链接的时候，应该会看到与图 8-11 类似的内容。在本范例的讨论部分中，会介绍一种使用 Boto（http://docs.pythonboto.org/en/latest/）自动创建策略的方式。

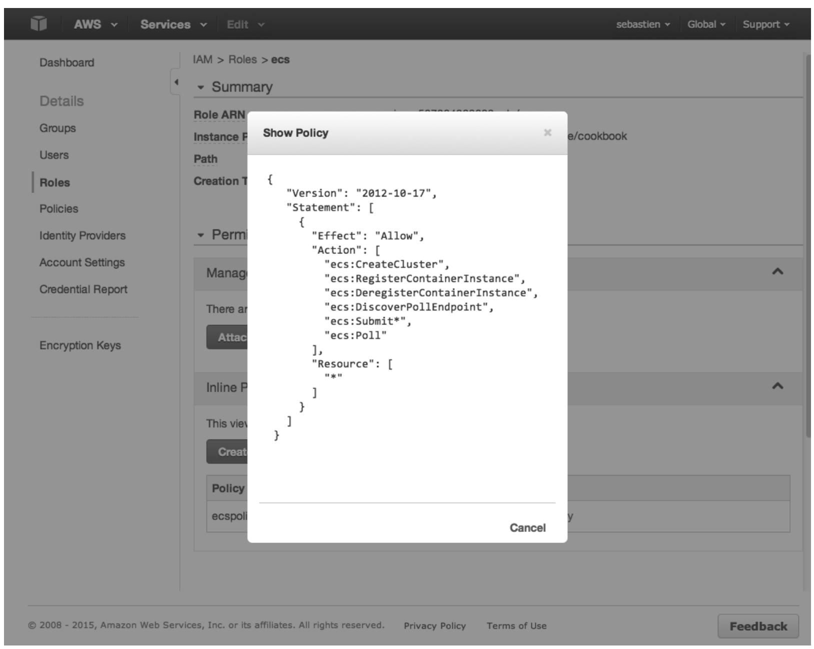


图 8-11：IAM 角色控制台中的 ECS 策略

(5) 安装最新版 AWS CLI（http://aws.amazon.com/cli/）。ECS API 的版本应该是 1.7.0 或者更高。你可以验证一下 `aws ecs` 命令是否可用，代码如下所示。

```
$ sudo pip install awscli
$ aws --version
aws-cli/1.7.8 Python/2.7.9 Darwin/12.6.0
$ aws ecs help

ECS()                                                              ECS()
```

```
NAME
       ecs -

DESCRIPTION
       Amazon  EC2  Container Service (Amazon ECS) is a highly scalable,
       fast, container management service that makes it easy to run,
       stop, and  manage  Docker containers on a cluster of Amazon
       EC2 instances. Amazon ECS lets you launch and stop
       container-enabled applications with simple API calls,  allows
       you to get the state of your cluster from a centralized service,
       and gives you access to many familiar Amazon EC2 features like
       security groups, Amazon EBS volumes, and IAM roles.
...
```

(6) 创建一个 AWS CLI 配置文件，这个配置文件包括你创建的 IAM 用户的 API 密钥信息。请注意，区域设置为 `us-east-1`，这是目前 ECS 可以使用的北弗吉尼亚地区，如下所示。

```
$ cat ~/.aws/config
[default]
output = table
region = us-east-1
aws_access_key_id = <your AWS access key>
aws_secret_access_key = <your AWS secret key>
```

完成了上面的所有步骤后，就可以开始使用 ECS 了。你需要创建一个集群（参见范例 8.12），定义与容器相关的任务，并在集群上运行这些任务来启动容器（参见范例 8.13）。

8.11.3　讨论

为了在集群上启动一个 ECS 实例，你需要创建一个 IAM 配置文件和 ECS 策略，如果你以前没有使用过 AWS，这也许会比较麻烦。为了帮助你完成这一步，你可以使用本书附带代码中的脚本，该脚本使用了 Python 的 Boto（http://docs.pythonboto.org/en/latest/）库来创建策略。

安装 Boto，将文件 /.aws/config 复制到 /.aws/credentials，克隆仓库然后执行脚本文件，如下所示。

```
$ git clone https://github.com/how2dock/docbook.git
$ sudo pip install boto
$ cp ~/.aws/config ~/.aws/credentials
$ cd ch08/ecs
$ ./ecs-policy.py
```

这段代码创建了一个 `ecs` 角色、一个 `ecspolicy` 的策略，以及一个 `cookbook` 的实例配置文件。你可以编辑这个脚本文件来修改这些名称。脚本执行完之后，你就可以在 IAM 管理控制台（https://console.aws.amacon.con/iam/home#roles）中看到刚创建的角色和策略了。

8.11.4　参考

- ECS 演示（https://aws.amazon.com/blogs/compute/amazon-ecs-video-demo/）视频

- ECS 官方文档（http://docs.aws.amazon.com/AmazonECS/latest/developerguide/get-set-up-for-amazon-ecs.html）

8.12 创建一个ECS集群

8.12.1 问题

你已经安装好了并准备使用 ECS（参见范例 8.11）。现在你打算在 ECS 上创建一个集群和一个在集群中运行容器的实例。

8.12.2 解决方案

使用在范例 8.11 中安装的 AWS CLI，并尝试一下新的 ECS API。在这个范例中，你将会学到如何使用下面的命令。

- `aws ecs list-clusters`
- `aws ecs create-cluster`
- `aws ecs describe-clusters`
- `aws ecs list-container-instances`
- `aws ecs delete-cluster`

默认情况下，你在 ECS 中拥有一个集群，但是直到你在其中启动一个实例为止，这个集群都处于非活动状态。尝试看一下下面集群的信息。

```
$ aws ecs describe-clusters
---------------------------------------------------------------------
|                         DescribeClusters                          |
+-------------------------------------------------------------------+
||                            failures                             ||
|+-------------------------------------------------------+----------+|
||                          arn                          | reason   ||
|+-------------------------------------------------------+----------+|
||  arn:aws:ecs:us-east-1:587534442583:cluster/default   |  MISSING ||
|+-------------------------------------------------------+-----------+
```

> 目前 AWS 限制每个用户只能使用两个 ECS 集群。

要想激活这个集群，可以通过 Boto 来启动一个实例。这里使用的是 ECS 专用的 AMI，这个 AMI 里面包括一个 ECS 代理（https://github.com/aws/amazon-ecs-agent）。要想通过 `ssh` 连接到 ECS 实例，你需要事先创建一个 SSH 密钥对，还需要一个拥有 ECS 策略的角色以及一个实例属性（实例配置文件，参见范例 8.11），如下所示。

```
$ python
...
>>> import boto
>>> c = boto.connect_ec2()
>>> c.run_instances('ami-34ddbe5c', \
                    key_name='ecs', \
                    instance_type='t2.micro', \
                    instance_profile_name='cookbook')
```

当一个实例启动后，等待它开始运行并注册到集群。然后可以再次查看集群状态。你将会看到默认的集群已经切换到了活动状态。你也可以列出所有容器实例，如下所示。

```
$ aws ecs describe-clusters
---------------------------------------------------------------------------------
|                              DescribeClusters                                 |
+-------------------------------------------------------------------------------+
||                                 clusters                                    ||
|+-----------------------------------------------------+--------------+--------+|
||                     clusterArn                       | clusterName  | status ||
|+-----------------------------------------------------+--------------+--------+|
||  arn:aws:ecs:us-east-1:587432148683:cluster/default |  default     | ACTIVE ||
|+-----------------------------------------------------+--------------+--------+|
$ aws ecs list-container-instances
--------------------------------------------------------------------------------
|                              ListContainerInstances                          |
+------------------------------------------------------------------------------+
||                               containerInstanceArns                        ||
|+----------------------------------------------------------------------------+|
||  arn:aws:ecs:us-east-1:541234428683:container-instance/ ...                ||
|+----------------------------------------------------------------------------+|
```

再启动另一个实例，增加集群的大小，如下所示。

```
$ aws ecs list-container-instances
--------------------------------------------------------------------------
|                                   ListContainerInstances               |
+------------------------------------------------------------------------+
||                                   containerInstanceArns              ||
|+----------------------------------------------------------------------+|
||  arn:aws:ecs:us-east-1:587342368683:container-instance/75738343-...  ||
||  arn:aws:ecs:us-east-1:587423448683:container-instance/b457e535-...  ||
||  arn:aws:ecs:us-east-1:584242468683:container-instance/e5c0be59-...  ||
||  arn:aws:ecs:us-east-1:587421468683:container-instance/e62d3d79-...  ||
|+----------------------------------------------------------------------+|
```

由于这些容器实例是通常的 EC2 实例，在 EC2 管理控制台中你也能看到这些实例。如果你已经设置好了 SSH 密钥，并且在安全组中打开了 22 端口，那么也能通过 ssh 连接到这些实例，如下所示。

```
$ ssh -i ~/.ssh/id_rsa_ecs ec2-user@52.1.224.245
...

    __|  __|  __|
    _|  (   \__ \  Amazon ECS-Optimized Amazon Linux AMI
```

```
    ____|\___|____/


    Image created: Thu Dec 18 01:39:14 UTC 2014
    PREVIEW AMI

  9 package(s) needed for security, out of 10 available
  Run "sudo yum update" to apply all updates.
  [ec2-user@ip-172-31-33-78 ~]$ docker ps
  CONTAINER ID    IMAGE                             ...
  4bc4d480a362    amazon/amazon-ecs-agent:latest     ...
  [ec2-user@ip-172-31-33-78 ~]$ docker version
  Client version: 1.6.2
  Client API version: 1.18
  Go version (client): go1.3.3
  Git commit (client): 7c8fca2/1.6.2
  OS/Arch (client): linux/amd64
  Server version: 1.6.2
  Server API version: 1.18
  Go version (server): go1.3.3
  Git commit (server): 7c8fca2/1.6.2
  OS/Arch (server): linux/amd64
```

可以看到，容器实例都运行着的 Docker 服务，ECS 代理也运行在容器中。你看到的 Docker 版本可能与上面的会不一样，因为 Docker 几乎每两个月就会发布一个新版本。

8.12.3　讨论

你可以使用默认的集群，也可以创建自己的集群，如下所示。

```
$ aws ecs create-cluster --cluster-name cookbook
---------------------------------------------------------------------------
|                            CreateCluster                                |
+-------------------------------------------------------------------------+
||                              cluster                                  ||
|+----------------------------------------------+--------------+----------+|
||                   clusterArn                | clusterName  | status   ||
|+----------------------------------------------+--------------+----------+|
||  arn:aws:ecs:us-east-1:...:cluster/cookbook |  cookbook    |  ACTIVE  ||
|+----------------------------------------------+--------------+----------+|
$ aws ecs list-clusters
-----------------------------------------------------------
|                     ListClusters                        |
+---------------------------------------------------------+
||                     clusterArns                       ||
|+-------------------------------------------------------+|
||  arn:aws:ecs:us-east-1:587264368683:cluster/cookbook  ||
||  arn:aws:ecs:us-east-1:587264368683:cluster/default   ||
|+-------------------------------------------------------+|
```

为了在新创建的集群中启动实例，你需要在创建实例的步骤中，设置一些用户数据（http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/user-data.html）。

如果使用 Boto，那么上面的工作可以通过下面的脚本来完成。

```
#!/usr/bin/env python

import boto
import base64

userdata="""
#!/bin/bash
echo ECS_CLUSTER=cookbook >> /etc/ecs/ecs.config
"""

c = boto.connect_ec2()
c.run_instances('ami-34ddbe5c', \
                key_name='ecs', \
                instance_type='t2.micro', \
                instance_profile_name='cookbook', \
                user_data=base64.b64encode(userdata))
```

当不再使用这个集群时，可以通过 `aws ecs delete-cluster --cluster cookbook` 命令来彻底删除这个集群。

8.12.4　参考

- ECS 代理在 GitHub 上的项目主页（https://github.com/aws/amazon-ecs-agent）

8.13　在ECS集群中启动Docker容器

8.13.1　问题

你知道如何在 AWS 上创建一个 ECS 集群（参见范例 8.12），现在你已经准备好在集群中的实例上运行 Docker 容器了。

8.13.2　解决方案

你可以通过一个 JSON 格式的定义文件来描述你要启动的容器或一组容器。这将被称为一个任务。你将会注册这个任务并运行它，这一过程分两步。当这个任务开始在集群中运行时，你可以对它进行 list、stop 和 start 等操作。

比如，要想使用 Docker Hub 上的 nginx 镜像在容器中运行 Nginx，你需要创建如下 JSON 格式的任务定义文件。

```
[
  {
    "environment": [],
    "name": "nginx",
    "image": "nginx",
    "cpu": 10,
    "portMappings": [
      {
```

```
        "containerPort": 80,
        "hostPort": 80
      }
    ],
    "memory": 10,
    "essential": true
  }
]
```

不难注意到，任务定义与 Kubernetes 中的 pod（参见范例 5.4）以及 Docker Compose 配置文件（参见范例 7.1）非常相似。

你可以使用 ECS 的 `register-task-definition` 命令来注册这个任务。也可以指定一个 family 参数来对任务进行分组，这还能帮助你保持版本历史记录，使用这些历史记录可以方便地进行回滚操作，如下所示。

```
$ aws ecs register-task-definition --family nginx \
                                   --cli-input-json file://$PWD/nginx.json
$ aws ecs list-task-definitions
------------------------------------------------------------------
|                       ListTaskDefinitions                      |
+----------------------------------------------------------------+
||                      taskDefinitionArns                      ||
|+--------------------------------------------------------------+|
||  arn:aws:ecs:us-east-1:584523528683:task-definition/nginx:1  ||
|+--------------------------------------------------------------+|
```

为了启动任务定义文件中定义的容器，你可以使用 `run-task` 命令，并指定你想要运行的容器的个数。要想停止容器，你需要停止相应的任务。要停止相应的任务，你需要指定该任务的 UUID，任务的 UUID 可以通过 `list-tasks` 命令获得，如下所示。

```
$ aws ecs run-task --task-definition nginx:1 --count 1
$ aws ecs stop-task --task 6223f2d3-3689-4b3b-a110-ea128350adb2
```

ECS 将任务分配到你的集群中的一个容器实例之上。镜像会从 Docker Hub 下载，容器使用任务定义文件里指定的参数启动。当前 ECS 还处于预览版阶段，因此找到一个任务所在的实例和为其分配的 IP 地址还都没有那么简单直接。如果你有多个实例在运行，还必须要进行一些猜测。看起来 ECS 中并不存在 Kubernetes 中类似的服务代理的组件。

8.13.3 讨论

在上面的 Nginx 例子中，一个任务在一个容器中运行，其实你也可以使用链接的容器定义一个任务。任务定义参考（http://docs.aws.amazon.com/AmazonECS/latest/developerguide/task_defintions.html）描述了所有可以在定义任务时使用的属性。作为示例，我们会在两个容器中运行一个 WordPress 服务（一个 wordpress 容器和一个 mysql 容器），你可以定义一个 wordpress 任务。AWS ECS 任务定义格式与 Docker Compose 的定义文件（参见范例 7.1）很相似。我们应该牢记，为 compose、pod 和 task 之间的标准化工作所付出的努力将会造福整个社区。

```
[
  {
    "image": "wordpress",
    "name": "wordpress",
    "cpu": 10,
    "memory": 200,
    "essential": true,
    "links": [
      "mysql"
    ],
    "portMappings": [
      {
        "containerPort": 80,
        "hostPort": 80
      }
    ],
    "environment": [
      {
        "name": "WORDPRESS_DB_NAME",
        "value": "wordpress"
      },
      {
        "name": "WORDPRESS_DB_USER",
        "value": "wordpress"
      },
      {
        "name": "WORDPRESS_DB_PASSWORD",
        "value": "wordpresspwd"
      }
    ]
  },
  {
    "image": "mysql",
    "name": "mysql",
    "cpu": 10,
    "memory": 200,
    "essential": true,
    "environment": [
      {
        "name": "MYSQL_ROOT_PASSWORD",
        "value": "wordpressdocker"
      },
      {
        "name": "MYSQL_DATABASE",
        "value": "wordpress"
      },
      {
        "name": "MYSQL_USER",
        "value": "wordpress"
      },
      {
        "name": "MYSQL_PASSWORD",
        "value": "wordpresspwd"
      }
    ]
```

```
  }
]
```

这个任务会像我们前面看到的 Nginx 的例子一样注册到集群，但是你指定了一个新的 family 参数。但是当这个任务开始执行的时候，可能会因为某些约束条件得不到满足而失败。在这个例子中，我的容器实例类型为 `t2.micro`，拥有 1 GB 的内存。由于任务定义中声明的 wordpress 和 mysql 容器各需要 500 MB 的内存，因此对集群调度器来说，没有足够内存来找到一个实例满足任务定义的约束条件，故任务运行会失败，如下所示。

```
$ aws ecs register-task-definition --family wordpress \
                                   --cli-input-json file://$PWD/wordpress.json
$ aws ecs run-task --task-definition wordpress:1 --count 1
--------------------------------------------------------------------------------
|                                  RunTask                                     |
+------------------------------------------------------------------------------+
||                                 failures                                   ||
|+---------------------------------------------------------+----------------+|
||                                arn                       |    reason      ||
|+---------------------------------------------------------+----------------+|
||  arn:aws:ecs:us-east-1:587264368683:container-instance/...|RESOURCE:MEMORY ||
||  arn:aws:ecs:us-east-1:587264368683:container-instance/...|RESOURCE:MEMORY ||
||  arn:aws:ecs:us-east-1:587264368683:container-instance/...|RESOURCE:MEMORY ||
|+----------------------------------------------------------------------------+|
```

你可以编辑任务定义文件，降低一些对内存的约束，然后使用相同的 family 注册一个新任务（这里版本为 2）。这次任务将会成功运行。如果登录到运行这个任务的实例中，你会发现这个任务的容器与 ECS 代理容器都在这个实例上运行，如下所示。

```
$ aws ecs run-task --task-definition wordpress:2 --count 1
$ ssh -i ~/.ssh/id_rsa_ecs ec2-user@54.152.108.134
...

    __|  __|  __|
    _|  (   \__ \  Amazon ECS-Optimized Amazon Linux AMI
  ____|\___|____/

...
[ec2-user@ip-172-31-36-83 ~]$ docker ps
CONTAINER ID  IMAGE              ... PORTS                       NAMES
36d590a206df  wordpress:4        ... 0.0.0.0:80->80/tcp          ecs-wordpress...
893d1bd24421  mysql:5            ... 3306/tcp                    ecs-wordpress...
81023576f81e  amazon/amazon-ecs ... 127.0.0.1:51678->51678/tcp  ecs-agent
```

享受 ECS 吧，并时刻关注 ECS 的改进以及公开发布。

8.13.4　参考

- 任务定义参考文档（http://docs.aws.amazon.com/AmazonECS/latest/developerguide/task_defintions.html）

8.14 利用AWS Beanstalk对Docker的支持在云中运行应用程序

8.14.1 问题

你希望在云中只需要上传 Dockerfile 就能部署一个基于 Docker 的应用程序。你希望云服务自动启动实例并对可能的负载平衡服务进行配置。

8.14.2 解决方案

使用 AWS Elastic Beanstalk（http://aws.amazon.com/elasticbeanstalk/）。Beanstalk 使用 AWS EC2 实例，可以自动创建弹性的负载平衡和安全组，并对你的应用程序和资源进行监控。Beanstalk 对 Docker 的支持发布于 2014 年 4 月（https://aws.amazon.com/blogs/aws/aws-elastic-beanstalk-for-docker/）。最初 Beanstalk 只支持单容器的应用，但是最近 AWS 发布了一个 AWS ECS 和 Beanstalk 的组合（http://docs.aws.amazon.com/elasticbeanstalk/latest/dg/create_deploy_docker_ecs.html）。这个组合允许 Beanstalk 将 ECS 集群当作应用程序的运行环境，并在一个实例中运行多个容器。

为了说明 Beanstalk 对 Docker 的支持，你将要通过 AWS CLI 工具构建一套 Beanstalk 环境，并通过单一的 Dockerfile 文件部署 2048 这个游戏（http://gabrielecirulli.github.io/2048/）。这个例子参考了官方 Beanstalk 文档（http://docs.aws.amazon.com/elasticbeanstalk/latest/dg/create_deploy_docker_image.html）。

在开始之前，你需要满足以下几个条件。

- 一个 AWS 账号（参见范例 8.1）
- AWS CLI（参见范例 8.2）
- 访问 Beanstalk 管理控制台（https://console.aws.amazon.com/elasticbeanstalk/）并根据屏幕指示注册 Beanstalk

部署应用程序需要下面三步。

(1) 使用 `awscli` 创建一个 Beanstalk 应用程序。
(2) 基于 Docker 软件栈创建一个 Beanstalk 环境。这个 Docker 软件栈在 Beanstalk 中被称为解决方案栈。
(3) 创建一个 Dockerfile 文件并通过 `eb` CLI 部署。

> 所有这些步骤都可以通过 AWS 管理控制台完成。本范例将会完全基于 CLI 进行部署，但是 `awscli` 调用的输出已进行删减。

使用 AWS CLI 创建一个名为 foobar 的应用程序，查看可用的解决方案栈，然后选择你需

要的 Docker 环境。使用你选择的解决方案栈创建一个配置文件模板，最后创建一个环境，如下所示。

```
$ aws elasticbeanstalk create-application --application-name foobar
...
$ aws elasticbeanstalk list-available-solution-stacks
...
$ aws elasticbeanstalk create-configuration-template
                        --application-name foobar
                        --solution-stack-name="64bit Amazon Linux 2014.09 v1.2.1 \
                        running Docker 1.5.0"
                        --template-name foo
...
$ aws elasticbeanstalk create-environment
                        --application-name foobar
                        --environment-name cookbook
                        --template-name foo
```

这时如果你打开 AWS Beanstalk 管理控制台页面，将会看到一个名为 foobar 的应用程序和名为 cookbook 的 Beanstalk 环境已经成功创建。

当创建完 Beanstalk 环境之后，你可以使用 `describe-environments` API 来确认 Beanstalk 环境是否已经创建完成。

在管理控制台中，你将会看到同时被创建的还有一个 EC2 主机实例、一个安全组以及一个弹性负载平衡。你可以通过 Beanstalk 管理控制台对负载平衡进行配置。

回到我们的 CLI 步骤，检查一下 Beanstalk 环境是否已经就绪，如下所示。

```
$ aws elasticbeanstalk describe-environments
--------------------------------------------------------------------------------
|                              DescribeEnvironments                            |
+------------------------------------------------------------------------------+
||                                Environments                                ||
|+------------------+---------------------------------------------------------+|
||  ApplicationName |  foobar                                                 ||
||  CNAME           |  cookbook-pmpgzmx2e6.elasticbeanstalk.com               ||
||  DateCreated     |  2015-03-30T15:32:47.814Z                               ||
||  DateUpdated     |  2015-03-30T15:38:14.291Z                               ||
||  EndpointURL     |  awseb-e-7-AWSEBLoa-CUXDVD6RL9R7-992275618.eu-west-1...||
||  EnvironmentId   |  e-7hamntqqnw                                           ||
||  EnvironmentName |  cookbook                                               ||
||  Health          |  Green                                                  ||
||  SolutionStackName|  64bit Amazon Linux 2014.09 v1.2.1 running Docker 1.5.0||
||  Status          |  Ready                                                  ||
|+------------------+---------------------------------------------------------+|
|||                                  Tier                                    |||
||+--------------------------------------+-----------------------------------+||
|||  Name                                |  WebServer                        |||
|||  Type                                |  Standard                         |||
|||  Version                             |                                   |||
||+--------------------------------------+-----------------------------------+||
```

当 Beanstalk 环境准备好之后，就可以将你的 Docker 应用程序推送到 Beanstalk 环境。这

可以通过 eb CLI 轻松完成，不过不幸的是，这个工具不包括在 awscli 中。为了完成部署，你需要执行下面的操作。

(1) 安装 awsebcli。
(2) 创建你的 Dockerfile 文件。
(3) 初始化之前创建的 foobar 应用程序。
(4) 查看 Beanstalk 环境，确认是否使用的是之前创建的 cookbook 环境。
(5) 部署这个应用程序。

让我们这就开始：安装 awsebcli，然后创建你的应用程序目录，并在这个目录下创建 Dockerfile 文件，如下所示。

```
$ sudo pip install awsebcli
$ mkdir beanstalk
$ cd beanstalk
$ cat > Dockerfile
FROM ubuntu:12.04

RUN apt-get update
RUN apt-get install -y nginx zip curl

RUN echo "daemon off;" >> /etc/nginx/nginx.conf
RUN curl -o /usr/share/nginx/www/master.zip -L https://codeload.github.com/ \
gabrielecirulli/2048/zip/master
RUN cd /usr/share/nginx/www/ && unzip master.zip && mv 2048-master/* . && \
    rm -rf 2048-master master.zip

EXPOSE 80

CMD ["/usr/sbin/nginx", "-c", "/etc/nginx/nginx.conf"]
```

然后你就可以使用 eb CLI 来初始化应用程序（使用上面步骤中的应用程序名 foobar），再使用 eb deploy 部署该应用程序，如下所示。

```
$ eb init foobar
$ eb list
* cookbook
$ eb deploy
Creating application version archive "app-150331_181300".
Uploading foobar/app-150331_181300.zip to S3. This may take a while.
Upload Complete.
INFO: Environment update is starting.
...
INFO: Successfully built aws_beanstalk/staging-app
INFO: Docker container ba7e79c37c43 is running aws_beanstalk/current-app.
INFO: New application version was deployed to running EC2 instances.
INFO: Environment update completed successfully.
```

现在你的应用程序已经部署完成。打开 Beanstalk 管理控制台（参见图 8-12），你会看到该应用程序的 URL，单击这个 URL 就可以打开 2048 游戏。在这个应用程序前面有一个弹性负载平衡，也就是说，随着游戏负载的增加，将会触发创建新实例的动作，该实例在负载平衡后面提供游戏应用。

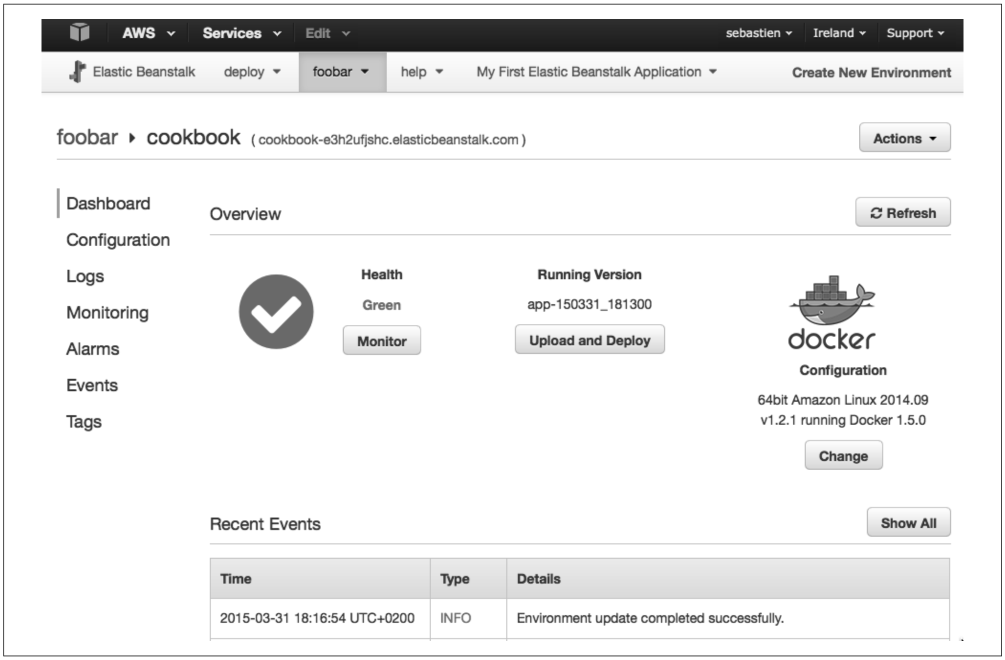


图 8-12：AWS Beanstalk 管理控制台

8.14.3 讨论

在以上例子中，我们的应用程序封装在了单一的 Dockerfile 文件中，且没有任何外部依赖。

第 9 章

监控容器

9.0 简介

当对分布式系统和分布式应用程序进行运维时，你希望能够获得尽可能详细的信息。你需要长期对大量的资源进行监控，预测成长趋势，并触发警报。你还需要收集在容器中运行的所有进程的日志并进行汇总，将数据进行持久化存储以便对这些日志做进一步的索引和搜索。最后，你还需要对所有这些信息进行可视化，以便快速浏览应用程序的状态，并在需要的时候进行调试。

本章首先会介绍一些基本的 Docker 命令，你可以在小规模部署环境下或者需要深入地对指定容器镜像研究时，使用这些命令进行基本的调试。范例 9.1 会介绍 `docker inspect` 命令，这个命令会返回给你所有关于指定容器的信息。范例 9.2 会告诉你怎么使用 `docker stats` 来获得关于特定容器的资源使用情况的流信息。最后，范例 9.3 将会介绍 `docker events` 命令，该命令会监听某一主机上的所有 Docker 事件。这些功能都可以通过 Docker 的 API 来实现，因此你可以通过支持这些 API 的任何 Docker 客户端来使用这些功能。

当构建应用程序时，你需要收集在容器中运行的服务的运行日志。这并不是什么特殊的监控内容，但是有时候你可以从日志中发现新的需要监控的指标。Docker 提供了一种简单的机制来查看在容器中运行的前台进程的 `stdout`，在范例 9.4 中会对此进行介绍。你也可以将这些日志重定向到远程 syslog 服务器，或者其他的日志聚合系统，例如 Fluentd，我们在范例 9.5 中会对此进行介绍。在日志记录驱动程序功能出现之前，有一个用于解决日志问题的容器方案备受瞩目，就是 `logspout`，在范例 9.6 中，我们会向你介绍 `logspout` 是如何工作的。尽管现在我们不再需要使用它了，但是作为一个有趣的软件，还是值得花些时间去了解一下。作为该节的结束部分，我们会在范例 9.8 中介绍如何在容器中部署 ELK 应用栈。ELK 是 Elasticsearch、Logstash 和 Kibana 的简称。Logstash 是一个日志聚合系统，可

以将数据传送到 Elasticsearch。Elasticsearch 是一个分布式数据存储，提供了有效的索引和搜索功能。Kibana 是一个仪表盘系统，对保存在 Elasticsearch 中的数据进行可视化。如果想要一个 ELK 的替代品，那么可能你会喜欢范例 9.11，此范例会以 InfluxDB（https://influxdb.com）作为数据存储，以 Grafana（http://grafana.org）作为仪表盘进行说明。

尽管 `docker stats` 为你提供了单一容器使用情况的统计镜像，但是你可能想收集多个容器的指标数据并对这些指标进行聚合。范例 9.9 是一个高级范例，涵盖了多个概念。它由两台主机组成，一台主机上运行着 ELK 应用栈，而另一台主机上运行着 `logspout` 和 `collectd`（https://collectd.org），这是一个用来收集系统统计信息的守护进程。在这个范例中，第二台主机上的 `collectd` 会通过 `docker stats` API 来收集所有在该主机上运行的容器的资源使用情况，并进一步通过 `logspout` 进行聚合，最后将这些数据发送到 ELK 主机上。如果你需要部署自己的监控解决方案，那么这部分内容非常值得一读。虽然上面的方案非常完美，不过我们还是会在范例 9.10 中介绍一下 cAdvisor，它是一个容器化（containerized）容器监控解决方案。你可以在所有 Docker 主机上部署 cAdvisor，它会监视所有在主机上运行的容器的所有资源使用情况。

在本章的最后，我们会在范例 9.12 中介绍一下 Weave Scope，这是一个容器基础设施的可视化工具。想象一下你的整个应用由数千个容器组成，拥有一个分布式应用的交互式管理器应该是非常具有吸引力的。Weave Scope 具备满足这一需求的潜力，它能让你对自己的应用程序有一个深入的理解。

9.1 使用docker inspect命令获取容器的详细信息

9.1.1 问题

你想获得关于某个容器的详细信息，比如，这个容器是什么时候创建的，运行的是什么命令，都有哪些端口映射，容器的 IP 地址是什么，等等。

9.1.2 解决方案

使用 `docker inspect` 命令。比如，启动一个 Nginx 容器并使用 `inspect`，如下所示。

```
$ docker run -d -p 80:80 nginx
$ docker inspect kickass_babbage
[{
    "AppArmorProfile": "",
    "Args": [
        "-g",
        "daemon off;"
    ],
...
        "ExposedPorts": {
            "443/tcp": {},
            "80/tcp": {}
        },
...
```

```
    "NetworkSettings": {
...
        "IPAddress": "172.17.0.3",
...
```

inspect 命令也可以对 Docker 镜像进行操作，如下所示。

```
$ docker inspect nginx
[{
    "Architecture": "amd64",
    "Author": "NGINX Docker Maintainers \"docker-maint@nginx.com\"",
    "Comment": "",
    "Config": {
        "AttachStderr": false,
        "AttachStdin": false,
        "AttachStdout": false,
        "Cmd": [
            "nginx",
            "-g",
            "daemon off;"
        ],
...
```

9.1.3 讨论

Docker 的 inspect 命令有一个 format 参数，你可以通过该参数指定一个 Golang 模板来获得一个容器或者镜像中指定的部分信息，而不是全部的信息的 JSON 输出，如下所示。

```
$ docker inspect --help

Usage: docker inspect [OPTIONS] CONTAINER|IMAGE [CONTAINER|IMAGE...]

Return low-level information on a container or image

  -f, --format=""    Format the output using the given go template.
  --help=false       Print usage
```

比如，可以这样获得一个运行中容器的 IP 地址，并检查其运行状态：

```
$ docker inspect -f '{{ .NetworkSettings.IPAddress }}' kickass_babbage
172.17.0.3
$ docker inspect -f '{{ .State.Running }}' kickass_babbage
true
```

如果你喜欢使用其他 Docker 客户端，比如 docker-py（参见范例 4.10），那么也可以通过标准 Python 字典类型（notation）来获取容器或者镜像的详细信息，如下所示。

```
$ python
...
>>> from docker import Client
>>> c=Client(base_url="unix://var/run/docker.sock")
>>> c.inspect_container('kickass_babbage')['State']['Running']
True
>>> c.inspect_container('kickass_babbage')['NetworkSettings']['IPAddress']
u'172.17.0.3'
```

9.2 获取运行中容器的使用统计信息

9.2.1 问题

你有一个在某一 Docker 主机上运行的容器，你想监控该容器的资源使用情况（比如内存、CPU 和网络）。

9.2.2 解决方案

使用 `docker stats` 命令。这个新 API 接口最初是 2015 年 2 月 10 日发布的，在 Docker 1.5 之后都可以使用。它的使用方法非常简单：指定容器名（或容器 ID），然后获取关于该容器统计信息的流输出。这里我们以启动一个 Flask 应用容器为例，对它调用 `stats` 方法，如下所示。

```
$ docker run -d -p 5000:5000 runseb/flask
$ docker stats dreamy_mccarthy
CONTAINER        CPU %  MEM USAGE/LIMIT       MEM %    NET I/O
dreamy_mccarthy  0.03%  24.01 MiB/1.955 GiB   1.20%    648 B/648 B
```

由于你接收到的是一个流输出，所以不能通过同时按 Ctrl+C 键来终止流。

9.2.3 讨论

通过命令行来快速获取容器的统计信息对交互式调试非常有用。但是，你应该会希望通过类似 Logstash（http://logstash.net）这样的日志收集系统来收集所有统计信息并进行聚合，以供之后的可视化和分析。

为了准备这样的监控框架，你可以使用 `curl` 命令通过 TCP 请求 Docker 守护进程来调用 Docker 的 `stats` API。

首先，你需要配置本地 Docker 守护进程让它监听 TCP 的 2375 端口。在 Ubuntu 系统上，可以编辑 /etc/default/docker 文件来确保该文件包含以下内容。

```
DOCKER_OPTS="-H tcp://127.0.0.1:2375"
```

然后，通过 `sudo service docker restart` 来重启 Docker 守护进程，现在你就可以使用 `curl` 来访问 Docker remote API 了。这个 API 的语法（http://docs.docker.com/engine/reference/api/docker_remote_api_v1.17/）同样非常简单：它是一个 HTTP GET 请求，URI 为 /containers/(id)/stats。我们可以像下面这样试一下。

```
$ $ docker -H tcp://127.0.0.1:2375 run -d -p 5001:5000 runseb/flask
$ curl http://127.0.0.1:2375/containers/agitated_albattani/stats
{"read":"2015-04-01T11:48:40.609469913Z",\
"network":{"rx_bytes":648,"rx_packets":8,"...
```

别忘了将 `agitated_albattani` 替换为你自己的容器的名称。之后你将会开始接收到统计信息的输出流，你可以通过 Ctrl+C 来中断接收流。考虑到印刷效果，我删除了上面命令输出的很多内容。如果你只是想尝试一下，那么这种方式非常方便，但是如果你喜欢 Python

（比如我），可能希望从 Python 程序中获取容器的统计信息。这可以通过使用 docker-py（参见范例 4.10）来实现。下面的 Python 脚本展示了该 API 的正确使用方式。

```
#!/usr/bin/env python

import json
import docker
import sys

cli=docker.Client(base_url='tcp://127.0.0.1:2375')
stats=cli.stats(sys.argv[1])
print json.dumps(json.loads(next(stats).rstrip('\n')),indent=4)
```

> Python 中的 `stats` 对象是一个 generator，它不像普通函数那样进行标准返回，而是一有返回结果就会对结果进行调用。它用来获取统计信息的输出流，并从上次中断的地方继续处理。脚本中的 `next(stats)` 方法用于从流中得到最新结果。

9.2.4 参考

- 最初 `stats` 在 GitHub 上的合并请求（https://github.com/docker/docker/pull/9984）
- API 文档（http://docs.docker.com/engine/reference/api/docker_remote_api_v1.17/）

9.3 在Docker主机上监听Docker事件

9.3.1 问题

你想监控主机上的 Docker 事件。你对取消镜像的标记、删除镜像以及容器的生命周期事件（比如创建、销毁和终止）感兴趣。

9.3.2 解决方案

使用 `docker events` 命令。该命令会返回你的 Docker 主机上发生的所有 Docker 事件流。该命令可以接受一些可选参数，比如，你可以设置只返回指定类型的事件，如下所示。

```
$ docker events --help

Usage: docker events [OPTIONS]

Get real time events from the server

  -f, --filter=[]    Provide filter values (i.e., 'event=stop')
  --help=false       Print usage
  --since=""         Show all events created since timestamp
  --until=""         Stream events until this timestamp
```

默认情况下，`docker events` 会持续运行并阻塞，直到你按下 Ctrl+C 来终止接收事件流，

不过你也可以像下面这样使用 --since 或者 --until 选项。

```
$ docker events --since 1427890602
2015-04-01T12:17:04....9393146cb55e5bc9f04e20eb5a0622b3e26aae7: untag
2015-04-01T12:17:09....d5266f8777bfba4974ac56e3270e7760f6f0a81: untag
2015-04-01T12:17:22....d5266f8777bfba4974ac56e3270e7760f6f0a85: untag
2015-04-01T12:17:23....66f8777bfba4974ac56e3270e7760f6f0a81253: delete
2015-04-01T12:17:23....e9b5a793523481a0a18645fc77ad53c4eadsfa2: delete
2015-04-01T12:17:23....878585defcc1bc6f79d2728a13957871b345345: delete
```

这只是一个提醒，你可以通过 date +"%s" 命令获取当前的时间戳。

9.3.3 讨论

Docker 也提供了与 events 命令相同的 API 调用，你可以使用 curl 命令来调用这个接口（参见范例 9.2）。让我们把这当作一个练习，并来看一个使用 docker-py 取得 Docker 事件列表的例子。

在范例 9.2 中，我们重新配置了 Docker 守护进程选项来启用基于 TCP 的远程 API。你也可以在 docker-py 中使用 Unix Docker socket。

下面这个 Python 脚本就是一个示例，它能省去你重新对 Docker 守护进程进行配置的麻烦。

```
#!/usr/bin/env python
import json
import docker
import sys

cli=docker.Client(base_url='unix://var/run/docker.sock')
events=cli.events(since=sys.argv[1],until=sys.argv[2])
for e in events:
    print e
```

这个脚本接受两个时间戳参数，并返回在这两个时间范围之内的 Docker 事件。下面是一个输出的例子。

```
$ ./events.py 1427890602 1427891476
{"status":"untag","id":"967a84db1eff36cab6e77fe9c9393146c...","time":1427890624}
{"status":"untag","id":"4986bf8c15363d1c5d15512d5266f8777...","time":1427890629}
{"status":"untag","id":"4986bf8c15363d1c5d15512d5266f8777...","time":1427890642}
{"status":"delete","id":"4986bf8c15363d1c5d15512d5266f877...","time":1427890643}
{"status":"delete","id":"ea13149945cb6b1e746bf28032f02e9b...","time":1427890643}
{"status":"delete","id":"df7546f9f060a2268024c8a230d86398...","time":1427890643}
```

一些诸如 StackStorm（http://stackstorm.com）等基于事件的工具充分利用了 Docker 事件功能，可以对基于 Docker 的基础设施的多个组成部分进行编排。

9.3.4 参考

- API 文档（http://docs.docker.com/engine/reference/api/docker_remote_api_v1.17/）

9.4 使用docker logs命令获取容器的日志

9.4.1 问题

你有一个运行中的容器，容器中有一个以前台方式运行的进程。你希望在该容器所在主机上访问这个进程的日志。

9.4.2 解决方案

使用 `docker logs` 命令。

举个例子，启动一个 Nginx 容器，并在浏览器上打开该主机的 80 端口，如下所示。

```
$ docker run -d -p 80:80 nginx
$ docker ps
CONTAINER ID    IMAGE           ...  PORTS                          NAMES
dd0e926c4015    nginx:latest    ...  443/tcp, 0.0.0.0:80->80/tcp    gloomy_mclean
$ docker logs gloomy_mclean
192.168.34.1 - - [10/Mar/2015:10:12:35 +0000] "GET / HTTP/1.1" 200 612 "-" ...
...
```

9.4.3 讨论

你可以通过 `-f` 选项获得一个持续输出的日志流，如下所示。

```
$ docker logs -f gloomy_mclean
192.168.34.1 - - [10/Mar/2015:10:12:35 +0000] "GET / HTTP/1.1" 200 612 "-" ...
...
```

此外，你可以通过 `docker top` 命令来监控容器内运行的进程，如下所示。

```
$ docker top gloomy_mclean
UID         PID         PPID   ...  CMD
root        5605        4732   ...  nginx: master process nginx -g daemon off;
syslog      5632        5605   ...  nginx: worker process
```

9.5 使用Docker守护进程之外的日志记录驱动程序

9.5.1 问题

默认情况下，Docker 通过 JSON 文件来保存容器的日志。你可以通过 `docker logs` 命令来查看容器的日志（参见范例 9.4）。但是你又想使用不同的方式来收集日志并进行聚合，你有可能会使用 `syslog` 或者 `journald` 这样的机制。

9.5.2 解决方案

启动容器时可以通过 `--log-driver` 选项指定一个日志记录驱动程序。这个功能是 Docker 1.6 才有的，并且随着新版本 Docker 的发布，又有一些日志记录驱动程序正在逐步添加进来。使用 Docker 的日志记录驱动程序功能，你可以将 Docker 容器日志重定向到 `syslog`、`journald`、`GELF`（Graylog Extended Log Format，https://www.graylog.org） 和 `Fluentd`（http://www.fluentd.org）。当然，你也可以通过将日志记录驱动程序设置为 `--log-driver=none` 来完全禁用日志记录功能。所有日志记录驱动程序及其配置选项都提供了详细的文档（https://docs.docker.com/reference/logging/overview/）。

> 如果你使用了默认 `json-file` 驱动程序之外的日志记录驱动程序，那么 `docker logs` 命令就不能工作了。

你可以使用日志记录驱动程序功能将你的日志重定向到本地 `syslog` 或 `journald`。但是为了以更高级的方式对日志记录驱动程序功能进行说明，我们这里会采用 Fluentd 来收集运行中容器的所有日志。首先，你需要在 Docker 主机上安装 Fluentd。最简单的安装方式就是使用 Treasure Data（http://www.treasuredata.com）提供的称为 td-agent 的 Fluentd 发行版。如果你相信它们的安装脚本，也可以通过 `curl` 命令来安装 td-agent，如下所示。

```
$ curl -L https://td-toolbelt.herokuapp.com/sh/\
          install-ubuntu-trusty-td-agent2.sh | sh
```

软件包安装完成后，你需要对 td-agent 进行配置，告诉它去匹配什么样的事件，并重定向到指定的位置。比如，要想匹配所有 Docker 事件（默认 Docker 事件会打上 docker.<CONTAINER_ID> 标签），并将它们重定向到标准输出，你需要编辑 td-agent 的配置文件 /etc/td-agent/td-agent.conf，添加下面的内容。

```
<match docker.**>
type stdout
</match>
```

然后重启 td-agent 服务，如下所示。

```
$ sudo service td-agent restart
```

现在你已经可以使用 Fluentd 来管理你的 Docker 日志了。让我们启动一个 Nginx 容器并使用这个日志记录驱动程序，如下所示。

```
$ docker run -d -p 80:80 --name nginx --log-driver=fluentd nginx
```

现在如果你从浏览器访问 Nginx 容器，然后去检查一下 td-agent 的日志文件，就会发现 Docker 容器的日志如下所示。

```
$ tail -n 3 /var/log/td-agent/td-agent.log
...
2015-08-17 13:41:10 docker.dc3a645abfaa: {"log":"192.168.33.1 ...,\
```

```
                                                    "container_id":"dc3a645abfaa...",\
                                                    "container_name":"/nginx",\
                                                    "source":"stdout"}
```

你会看到，Docker 的日志都被加上了 `docker.<CONTAINER_ID>` 前缀。如果想为 Docker 日志设置其他前缀，你可以传递一个 Go 模板参数（现在支持 `{{.ID}}`、`{{.FullID}}`、`{{.Name}}`）。比如，你想将 Docker 日志的前缀设置为容器名称，则可以像下面这样来设置 `log-opt` 选项。

```
$ docker kill nginx
$ docker rm nginx
$ docker run -d -p 80:80 --name nginx \
                       --log-driver=fluentd \
                       --log-opt fluentd-tag=docker.{{.Name}} nginx
```

新的日志将会像下面这样。

```
$ tail -n 3 /var/log/td-agent/td-agent.log
...
2015-08-17 13:43:45 docker./nginx: {"container_id":"e4152ad9bdba...",\
                                    "container_name":"/stupefied_franklin",\
                                    "source":"stdout",\
                                    "log":"192.168.33.1 ...}
```

在这个例子中，你只是将 Docker 容器的日志重定向到了 Fluentd 自己的日志，在实际环境中并没有多大用处。在部署生产环境时，你可能会将日志重定向到远程数据存储，比如 `elasticsearch`、`influxdb` 或 `mongoDB`。

9.5.3　讨论

在解决方案部分，我们在 Docker 主机上以本地服务方式启动了 td-agent。你也可以在本地以容器方式来运行它。在你的当前工作目录下，创建一个名为 test.conf 的配置文件，其内容如下所示。

```
<source>
  type forward
</source>
<match docker.**>
  type stdout
</match>
```

接着让我们启动 fluentd 容器。你需要指定一个卷来将本地配置文件挂载到运行的容器中，并通过设置一个环境变量指向该文件，如下所示。

```
$ docker run -it -d -p 24224:24224 -v /path/to/conf:/fluentd/etc \
-e FLUENTD_CONF=test.conf fluent/fluentd:latest
```

默认配置下，`fluentd` 日志记录驱动程序会连接在本地主机上监听 24224 端口的 `fluentd` 服务器。因此，当你以 `--log-driver=fluentd` 选项启动其他容器时，该容器会自动连接到在上面容器中运行的 `fluentd`。

像前面的例子一样，我们现在启动一个 Nginx 容器，并使用 `docker logs` 命令查看 Fluentd 容器的日志。

9.5.4 参考

- 配置 Docker 日志记录驱动程序（https://docs.docker.com/reference/logging/overview/）
- Docker 的 Fluentd 日志记录驱动程序文档（https://github.com/docker/docker/blob/master/docs/reference/logging/fluentd.md）

9.6 使用Logspout采集容器日志

9.6.1 问题

在范例 9.4 中我们已经看到，容器的日志可以通过 `docker logs` 命令来查看，但是你想从在不同 Docker 主机上运行的容器中收集日志并进行聚合。

9.6.2 解决方案

使用 `logspout`（https://github.com/gliderlabs/logspout）。Logspout 可以收集一台 Docker 主机上的所有容器的日志并把它们路由到其他主机。它以容器的方式运行，并且是完全无状态的。

你可以使用 Logspout 将日志路由到一台 syslog 服务器或者发送到 Logstash（http://logstash.net）来处理。Logspout 诞生于 Docker 1.6 之前，也正是在这个版本中，Docker 引入了日志记录驱动程序（参见范例 9.5）功能。现在你仍然可以使用 Logspout，但是日志记录驱动程序功能也为你提供了一种进行日志重定向的简单选择。

让我们在一台 Docker 主机上安装 Logspout，从一个 Nginx 容器收集日志。你的 `nginx` 在宿主机的 80 端口上运行。启动 `logspout`，将宿主机的 Docker Unix socket/var/run/docker.sock 挂载到 logspout 容器的 /tmp/docker.sock 上，并指定一个 syslog 地址（这里使用了另一台 IP 地址为 192.168.34.11 的 Docker 主机），如下所示。

```
$ docker pull nginx
$ docker pull gliderlabs/logspout
$ docker run -d --name webserver -p 80:80 nginx
$ docker run -d --name logspout -v /var/run/docker.sock:/tmp/docker.sock \
             gliderlabs/logspout syslog://192.168.34.11:5000
```

为了收集日志，你需要在 192.168.34.11 上运行 Logstash 容器。为方便起见，它将会在 UDP 5000 端口上监听 syslog 输入，并全部输出到同一台主机的标准输出中。先从拉取 logstash 镜像开始（在这个例子中使用了 ehazlett/logstash 镜像，但是还有很多其他 Logstash 镜像供你选择）。镜像下载完成之后，你需要构建自己的 Logstash 镜像并使用自定义 Logstash 配置文件（基于 ehazlett/logstash 镜像的 /etc/logstash.conf.sample），如下所示。

```
$ docker pull ehazlett/logstash
$ cat logstash.conf
input {
  tcp {
    port => 5000
    type => syslog
```

```
  }
}

filter {
  if [type] == "syslog" {
    grok {
      match => { "message" => "%{SYSLOGTIMESTAMP:syslog_timestamp} \
      %{SYSLOGHOST:syslog_hostname} \
      %{DATA:syslog_program}(?:\[%{POSINT:syslog_pid}\])?: \
      %{GREEDYDATA:syslog_message}" }
      add_field => [ "received_at", "%{@timestamp}" ]
      add_field => [ "received_from", "%{host}" ]
    }
    syslog_pri { }
    date {
      match => [ "syslog_timestamp", "MMM  d HH:mm:ss", "MMM dd HH:mm:ss" ]
    }
  }
}

output {
  stdout { codec => rubydebug }
}
$ cat Dockerfile
FROM ehazlett/logstash

COPY logstash.conf /etc/logstash.conf
ENTRYPOINT ["/opt/logstash/bin/logstash"]
$ docker build -t logstash .
```

现在就可以启动 Logstash 容器了，并将容器的 5000 端口绑定到宿主机的 5000 端口，监听 UDP 协议的通信，如下所示。

```
$ docker run -d --name logstash -p 5000:5000/udp logstash -f /etc/logstash.conf
```

如果打开浏览器访问第一个 Docker 主机上运行的 Nginx，Logstash 容器就能看到相应的日志输出，如下所示。

```
$ docker logs logstash
...
{
  "message" => "<14>2015-03-10T13:00:39Z 889bbf0753a8 nginx[1]: 192.168.34.1 - \
              - [10/Mar/2015:13:00:39 +0000] \"GET / HTTP/1.1\" 200 612 \"-\"
                 \"Mozilla/5.0 \
              (Macintosh; Intel Mac OS X 10_8_5) \
              AppleWebKit/600.3.18 (KHTML, like Gecko) \
              Version/6.2.3 Safari/537.85.12\" \"-\"\n",
              "@version" => "1",
              "@timestamp" => "2015-03-10T13:00:36.241Z",
                   "type" => "syslog",
                   "host" => "192.168.34.10",
                   "tags" => [
...
```

9.6.3 讨论

为了简化使用 Logstash 对 Logspout 进行的测试，你可以克隆本书附带的仓库，切换到 ch09/logspout 目录。该目录下的 Vagrantfile 文件会启动两台 Docker 主机，并在每台主机上拉取所需要的 Docker 镜像，如下所示。

```
$ git clone https://github.com/how2dock/docbook.git
$ cd ch09/logspout
$ vagrant up
$ vagrant status
Current machine states:

w                          running (virtualbox)
elk                        running (virtualbox)
...
```

在 web server 节点上，你可以运行 Nginx 和 Logspout 容器。在 elk 节点上，你可以运行 Logstash 容器，如下所示。

```
$ vagrant ssh w
$ docker run --name nginx -d -p 80:80 nginx
$ docker run -d --name logspout -v /var/run/docker.sock:/tmp/docker.sock \
             gliderlabs/logspout syslog://192.168.34.11:5000

$ vagrant ssh elk
$ cd /vagrant
$ docker build -t logstash .
$ docker run -d --name logstash -p 5000:5000/udp logstash -f /etc/logstash.conf
```

你应该会在 Logstash 容器中看到 Nginx 的日志。可以试一下使用更多的主机和不同的容器，并使用 Logstash 插件将你的日志以不同的格式保存。

9.6.4 参考

- Logstash 官网（http://logstash.net）
- Logstash 配置说明（http://logstash.net/docs/1.4.2/configuration）
- 用于 Logstash 输入、输出、编码解码和过滤的插件（http://logstash.net/docs/1.4.2/index）

9.7 管理Logspout路由来存储容器日志

9.7.1 问题

你正在使用 Logspout 将日志流发送到远程服务器，但是你想修改远程服务器的 URI。特别是你想通过直接在 Logspout 中查看日志来对容器进行调试，修改路由的 URI，或者添加更多的路由 URI。

9.7.2　解决方案

在范例 9.6 中，你可能已经注意到了 Logspout 容器暴露了 8000 端口。你可以使用这个端口，通过简单的 HTTP API 来管理路由。

你可以将 8000 端口绑定到宿主机上，然后远程访问 API，但是作为练习，你要在本地使用链接的容器来实现这一功能。拉取一个带有 `curl` 命令的镜像，然后启动一个交互式容器。确认你能 ping 通 Logspout 容器（这里假设你采用与范例 9.6 相同的设置）。然后通过 `curl` 来访问位于 http://logspout:8000 的 Logspout API。

```
$ docker pull tutum/curl
$ docker run -ti --link logspout:logspout tutum/curl /bin/bash
root@c94a4eacb7cc:/# ping logspout
PING logspout (172.17.0.10) 56(84) bytes of data.
64 bytes from logspout (172.17.0.10): icmp_seq=1 ttl=64 time=0.075 ms
...
root@c94a4eacb7cc:/# curl http://logspout:8000/logs
        logspout|[martini] Started GET /logs for 172.17.0.12:38353
        nginx|192.168.34.1 [10/Mar/2015:13:57:38 +0000] "GET / HTTP/1.1" 200 ...
        nginx|192.168.34.1 [10/Mar/2015:13:57:43 +0000] "GET / HTTP/1.1" 200 ...
```

9.7.3　讨论

为了管理日志流，API 公开了一个 /routes 路由。HTTP 动作 `GET`、`DELETE` 和 `POST` 分别用来获取、删除和更新日志流路由，如下所示。

```
root@1fbb2f9636a8:/# curl http://logspout:8000/routes
[
  {
    "id": "e508de0c9689",
    "target": {
      "type": "syslog",
      "addr": "192.168.34.11:5000"
    }
  }
]
root@1fbb2f9636a8:/# curl http://logspout:8000/routes/e508de0c9689
{
  "id": "e508de0c9689",
  "target": {
    "type": "syslog",
    "addr": "192.168.34.11:5000"
  }
}
root@1fbb2f9636a8:/# curl -X DELETE http://logspout:8000/routes/e508de0c9689
root@1fbb2f9636a8:/# curl http://logspout:8000/routes
[]
root@1fbb2f9636a8:/# curl -X POST \
                          -d ‘{"target": {"type": "syslog", \
                                          "addr": "192.168.34.11:5000"}}' \
                          http://logspout:8000/routes
{
```

```
    "id": "f60d30502654",
    "target": {
      "type": "syslog",
      "addr": "192.168.34.11:5000"
    }
  }
root@1fbb2f9636a8:/# curl http://logspout:8000/routes
[
  {
    "id": "f60d30502654",
    "target": {
      "type": "syslog",
      "addr": "192.168.34.11:5000"
    }
  }
]
```

你可以创建一个指向 Papertrail（https://papertrailapp.com）的路由，自动将日志备份到 Amazon S3。

9.8 使用Elasticsearch和Kibana对容器日志进行存储和可视化

9.8.1 问题

范例 9.6 使用 Logstash（http://logstash.net）来接收日志并将日志发送到标准输出。但是 Logstash 有很多插件（http://logstash.net/docs/1.4.2/index）可以让你进行更多的处理。你可能想更进一步，使用 Elasticsearch（http://www.elasticsearch.com）来存储容器的日志。

9.8.2 解决方案

启动一个 Elasticsearch 和 Kibana 容器。Kibana（http://www.elasticsearch.org/overview/kibana/）是一个仪表盘系统，允许你查询 Elasticsearch 中的索引并方便进行可视化。使用 ehazlett/logstash 镜像以及它的默认配置启动一个 Logstash 容器，如下所示。

```
$ docker run --name es -d -p 9200:9200 -p 9300:9300 ehazlett/elasticsearch
$ docker run --name kibana -d -p 80:80 ehazlett/kibana
$ docker run -d --name logstash -p 5000:5000/udp \
             --link es:elasticsearch ehazlett/logstash \
             -f /etc/logstash.conf.sample
```

注意，Logstash 容器链接到了 Elasticsearch 容器。如果你不进行容器链接，Logstash 将找不到 Elasticsearch 服务。

在容器启动之后，你可以打开浏览器访问 Kibana 容器所在 Docker 主机的 80 端口。你将会看到 Kibana 默认的仪表盘。选择提供的示例仪表盘可以查看你的索引的信息，并创建一个基本的仪表盘。你可以看到访问 Nginx 服务器所产生的日志，如图 9-1 所示。

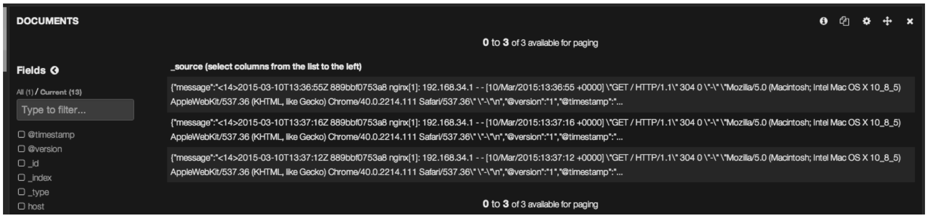


图 9-1：Kibana 仪表盘快照

9.8.3 讨论

在这个解决方案中，Elasticsearch 在独立的容器中运行。如果你停止并删除了 Elasticsearch 容器，那么为存储 Logspout 采集的日志流而创建的索引信息将会丢失。可以考虑挂载一个本地卷备份索引以对 Elasticsearch 的数据进行持久化。另外，如果想要更大的存储和高效的索引，你可以考率创建一个跨越多台 Docker 主机的 Elasticsearch 集群。

9.9 使用Collectd对容器指标进行可视化

9.9.1 问题

除了对应用程序的日志进行可视化（参见范例 9.8）之外，你可能还会喜欢监控容器的各种指标，比如 CPU。

9.9.2 解决方案

使用 Collectd（https://collectd.org）。在所有你希望对容器进行监控的主机上运行该容器。将 /var/run/docker.sock 文件挂载到名为 collectd 的容器中，你可以使用 Collectd 插件通过 Docker 统计 API（参见范例 9.2）采集指标数据，并将指标数据发送到在其他主机上运行的 Graphite 仪表盘中。

> 这是一个高级范例，使用了前面介绍过的一些概念。请确认在阅读本范例之前你已经阅读过范例 7.1 和范例 9.8。

为了进行测试，你需要使用下面由两台 Docker 主机构成的配置。一台主机上运行四个容器：一个 Nginx 容器用来向标准输出输出测试用日志，一个 Logspout 容器用于将所有标准输出日志发送到 Logstash 实例，一个容器用于产生模拟负载（比如 borja/unixbench），以及

一个 Collectd 容器。

另一台 Docker 主机上也运行着四个容器：一个 Logstash 容器用于接收来自 Logspout 的日志，一个 Elasticsearch 容器用于存储日志，一个 Kibana 容器用于对日志进行可视化，以及一个 Graphite 容器。Graphite 容器里也运行着 carbon 来存储指标数据。

图 9-2 显示了这种双主机八容器设置。

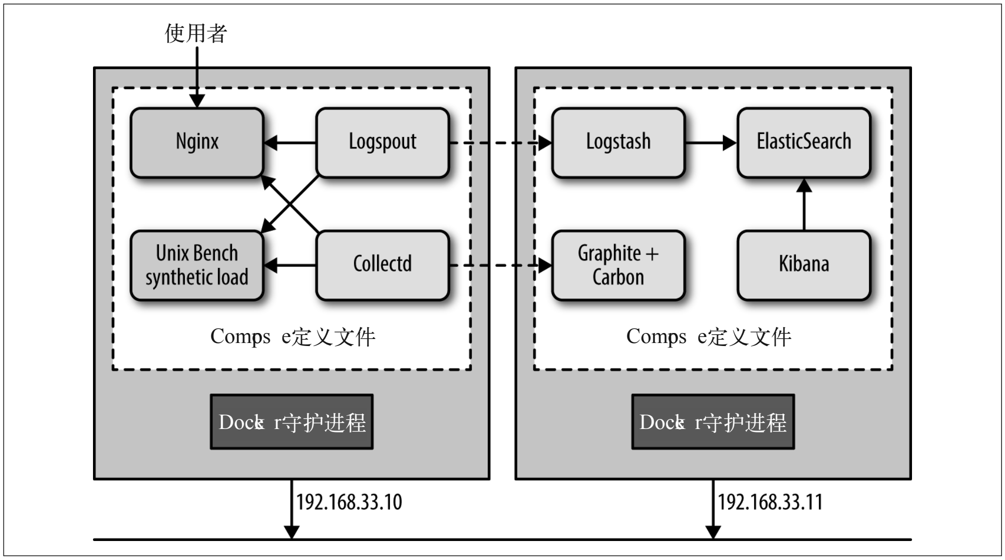


图 9-2：双主机、Collectd、Logstash、Kibana 和 Graphite 设置

在第一台主机（工作者）上，你可以通过 Docker Compose（参见范例 7.1）使用如下所示的 YAML 文件来启动所有容器，如下所示。

```
nginx:
  image: nginx
  ports:
   - 80:80
logspout:
  image: gliderlabs/logspout
  volumes:
   - /var/run/docker.sock:/tmp/docker.sock
  command: syslog://192.168.33.11:5000
collectd:
  build: .
  volumes:
   - /var/run/docker.sock:/var/run/docker.sock
load:
  image: borja/unixbench
```

Logspout 容器使用 command 参数指定你的 Logstash 端点。如果你在不同的网络环境中运行，则需要修改上面的 Logstash IP 地址。Collectd 容器由 Docker Compose 构建，其 Dockerfile 如下所示。

```
FROM debian:jessie

RUN apt-get update && apt-get -y install \
    collectd \
    python \
    python-pip
RUN apt-get clean
RUN pip install docker-py

RUN groupadd -r docker && useradd -r -g docker docker

ADD docker-stats.py /opt/collectd/bin/docker-stats.py
ADD docker-report.py /opt/collectd/bin/docker-report.py
ADD collectd.conf /etc/collectd/collectd.conf

RUN chown -R docker /opt/collectd/bin

CMD ["/usr/sbin/collectd","-f"]
```

在本范例的讨论部分，你还会再次重温这个 Dockerfile 中用到的脚本。

在第二台主机（监控者）上，可以通过 Docker Compose（参见范例 7.1）使用如下所示的 YAML 文件来启动所有容器。

```
es:
  image: ehazlett/elasticsearch
  ports:
   - 9300:9300
   - 9200:9200
kibana:
  image: ehazlett/kibana
  ports:
   - 8080:80
graphite:
  image: hopsoft/graphite-statsd
  ports:
   - 80:80
   - 2003:2003
   - 8125:8125/udp
logstash:
  image: ehazlett/logstash
  ports:
   - 5000:5000
   - 5000:5000/udp
  volumes:
   - /root/docbook/ch09/collectd/logstash.conf:/etc/logstash.conf
  links:
   - es:elasticsearch
  command: -f /etc/logstash.conf
```

在这个例子中，我们使用了几个非官方的镜像：gliderlabs/logspout、borja/unixbench、ehazlett/elasticsearch、ehazlett/kibana、ehazlett/logstash 和 hopsoft/graphite-statsd。你可以在 Docker Hub 上查看这些镜像的 Dockerfile，如果不信任这些镜像，也可以自己构建。

当两台主机上的容器都启动之后，并假设你正确设置了网络和防火墙（如果你正在使用云主机实例，还需要打开指定安全组的端口），就能够访问到工作者主机 80 端口上的 Nginx 容器，监控主机 8080 端口上的 Kibana 仪表板以及 80 端口上的 Graphite 仪表盘。

Graphite 仪表盘将会显示来自所有在工作者主机上运行的容器的基本 CPU 指标。你将会看到与图 9-3 类似的内容。

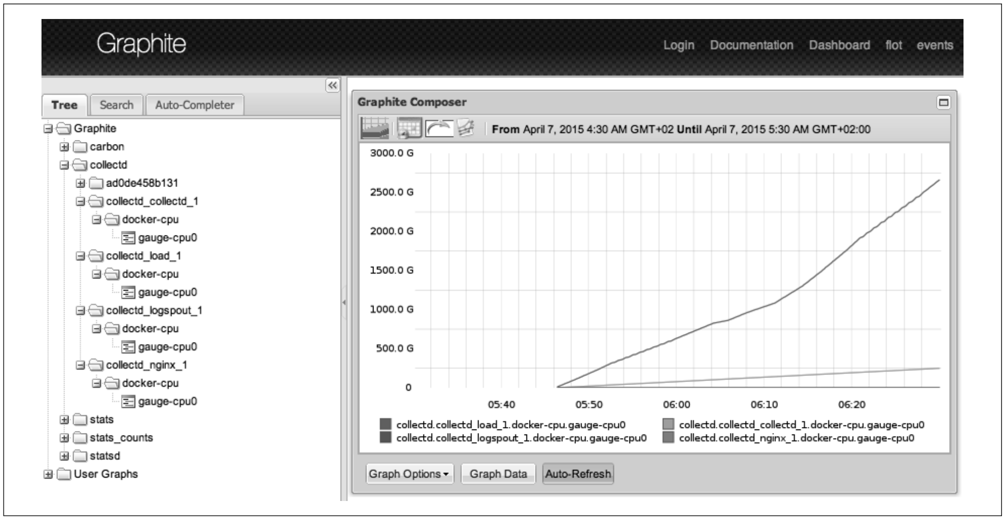


图 9-3：Graphite 仪表盘显示了来自所有容器的 CPU 指标

9.9.3 讨论

你可以通过使用本书随附的在线材料，获取范例中使用的全部脚本。如果之前还没有克隆过这个仓库，你需要先克隆它，然后切换到 docbook/ch09/collectd 目录下，如下所示。

```
$ git clone https://github.com/how2dock/docbook.git
$ cd docbook/ch09/collectd
$ tree
.
├── Dockerfile
├── README.md
├── Vagrantfile
├── collectd.conf
├── docker-report.py
├── docker-stats.py
├── logstash.conf
├── monitor.yml
└── worker.yml
```

这个 Vagrantfile 会在你的本地主机上启动两台 Docker 主机，以进行上面的实验。但是你也可以将这个仓库克隆到两台安装了 Docker 和 Docker Compose 的云主机实例中，并启动所有容器。如果你正在使用 Vagrant，请根据下面内容进行操作。

```
$ vagrant up
$ vagrant ssh monitor
$ vagrant ssh worker
```

> 在本范例中使用 Vagrant 进行实验时，我遇到了几次随机错误，以及镜像下载缓慢的问题。如果使用具有更好网络连通性的云主机实例，可能体验效果会更好。

这两个 YAML 用于在两台主机上方便地启动所有容器。请不要在同一台主机上运行这两条命令，如下所示。

```
$ docker-compose -f monitor.yml up -d
$ docker-compose -f worker.yml up -d
```

logstash.conf 文件已经在范例 9.6 中讨论过了。如果你还不理解这个配置文件，请返回该范例再次阅读一下。

Dockerfile 文件用于构建 Collectd 镜像，如前面解决方案部分所述。该镜像基于 Debian Jessie 镜像，并安装了 docker-py（参见范例 4.10）以及一些其他脚本。

Collectd 使用插件（https://collectd.org/wiki/index.php/Table_of_Plugins）来收集指标数据并将它们发送到数据存储（比如 Graphite 的 Carbon）。在这个例子中，我们使用了最简单的 Collectd 插件，该插件名为 `exec`。该插件在 collectd.conf 文件中进行定义，如下所示。

```
<Plugin exec>
   Exec "docker" "/opt/collectd/bin/docker-stats.py"
   NotificationExec "docker" "/opt/collectd/bin/docker-report.py"
</Plugin>
```

Collectd 容器中的 Collectd 进程以前台方式运行，定期执行配置文件中设置的两个 Python 脚本。这也是你在 Dockerfile 中复制这两个文件的原因。docker-report.py 脚本会将值输出到 syslog。这样做带来的好处是，你还可以通过 Logspout 容器来收集这些数据并在 Kibana 仪表盘中进行展示。docker-stats.py 脚本使用了 Docker stats API（参见范例 9.2）和 `docker-py` Python 包。这个脚本会列出所有正在运行的容器，并获取这些容器的统计信息。以名为 `cpu_stats` 的统计信息为例，这个脚本会向标准输出写入一个 PUTVAL 字符串。Collectd 会解析这个字符串并发送到 Graphite 数据存储（又名 Carbon）进行存储和可视化。这个 PUTVAL 字符串遵循 Collectd `exec` 插件的语法，如下所示。

```
#!/usr/bin/env python

import random
import json
import docker
import sys

cli=docker.Client(base_url='unix://var/run/docker.sock')

types = ["gauge-cpu0"]
```

```
for h in cli.containers():
    if not h["Status"].startswith("Up"):
        continue
    stats = json.loads(cli.stats(h["Id"]).next())
    for k, v in stats.items():
        if k == "cpu_stats":
            print("PUTVAL %s/%s/%s N:%s" % (h['Names'][0].lstrip('/'), \
                                'docker-cpu', types[0], \
                                v['cpu_usage']['total_usage']))
```

本范例中的示例插件是一个非常简单的插件，实际上对容器的统计数据也需要进一步进行加工处理。你可能会考虑使用这个（https://github.com/cloudwatt/docker-collectd-plugin）基于 Python 的插件替代。

9.9.4 参考

- Collectd 官方网站（https://collectd.org）
- Collectd Exec 插件（http://collectd.org/documentation/manpages/collectd-exec.5.shtml）
- Graphite 官方网站（http://graphite.wikidot.com）
- Logstash 官方网站（http://logstash.net）
- Collectd Docker 插件（https://github.com/cloudwatt/docker-collectd-plugin）

9.10 使用cAdvisor监控容器资源使用状况

9.10.1 问题

尽管使用 Logspout（参见范例 9.6）可以将应用程序的日志发送到远程端点，但是你需要一个资源利用率监控系统。

9.10.2 解决方案

使用 cAdvisor（https://github.com/google/cadvisor），这是一个 Google 创建的用来监控他们的 lcmtfy（https://github.com/google/lmctfy）容器资源使用情况和性能的软件。cAdvisor 在你的 Docker 主机上以容器方式运行。通过挂载本地卷，它可以监控在同一台主机上运行的所有容器。它还提供了一个本地 Web 界面和 API（https://github.com/google/cadvisor/blob/master/docs/api.md），并且能将数据存储到 InfluxDB（http://influxdb.com）。将运行中容器的数据存储到远程 InfluxDB 集群，这样你就可以对所有在一个集群中运行的容器的性能指标进行聚合。

开始我们先使用一台主机。下载 cAdvisor 镜像和 borja/unixbench 镜像，borja/unixbench 镜像可以让你在一个容器中模拟产生工作负载。

```
$ docker pull google/cadvisor:latest
$ docker pull borja/unixbench
```

```
$ docker run -v /var/run:/var/run:rw\
             -v /sys:/sys:ro \
             -v /var/lib/docker/:/var/lib/docker:ro \
             -p 8080:8080 \
             -d \
             --name cadvisor \
             google/cadvisor:latest
$ docker run -d borja/unixbench
```

当两个容器启动后，你可以打开浏览器访问 http://<IP_DOCKER_HOST>:8080 使用 cAdvisor UI（参见图 9-4）。你将会看到正在运行的容器以及每个容器的指标数据。

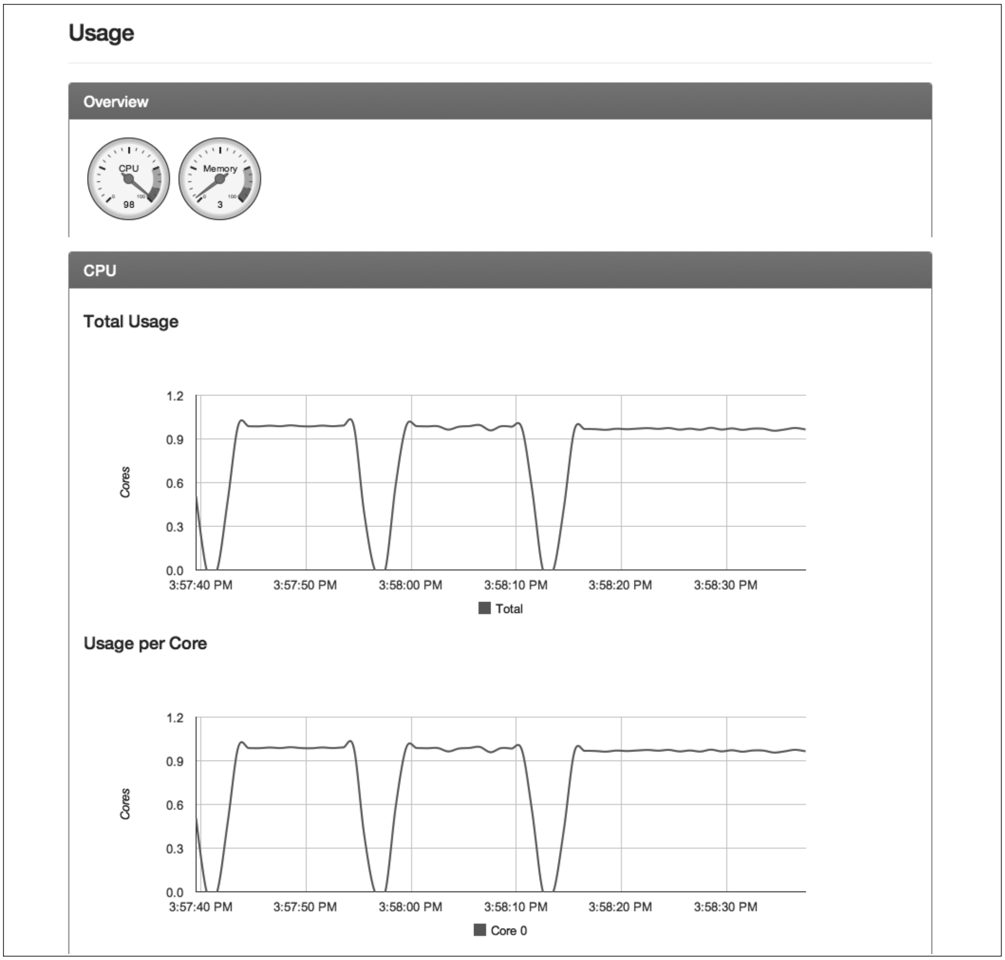


图 9-4：cAdvisor UI

9.10.3　参考

- cAdvisor API 文档（https://github.com/google/cadvisor/blob/master/docs/api.md）

9.11 通过InfluxDB、Grafana和cAdvisor监控容器指标

9.11.1 问题

你想在日志采集和性能监控技术栈中使用 Elastic/Logstash/Kibana 之外的替代方案。

9.11.2 解决方案

考虑使用 cAdvisor（参见范例 9.10）与 InfluxDB（https://influxdb.com）的组合存储时间序列数据，使用 Grafana（http://grafana.org）对数据进行可视化。cAdvisor 从所有在 Docker 主机上运行的容器采集数据指标，它的 InfluxDB 存储驱动程序允许你将所有指标数据作为时间序列存储到 InfluxDB（一个用于存储时间序列数据的分布式数据库）。Grafana 相当于 Kibana，对来自 InfluxDB 的数据进行可视化。

下面是单节点的基本配置。你将运行 cAdvisor，并将它配置为将数据发送到一台 InfluxDB 主机，同时你也会运行 InfluxDB 和 Grafana。所有这些服务都在容器中运行，如下所示。

```
$ docker run -d -p 8083:8083 -p 8086:8086 \
             -e PRE_CREATE_DB="db" \
             --name influxdb \
             tutum/influxdb:0.8.8
$ docker run -d -p 80:80 \
             --link=influxdb:influxdb \
             -e HTTP_USER=admin \
             -e HTTP_PASS=root \
             -e INFLUXDB_HOST=influxdb \
             -e INFLUXDB_NAME=db \
             --name=grafana \
             tutum/grafana
$ docker run -v /var/run:/var/run:rw \
             -v /sys:/sys:ro \
             -v /var/lib/docker/:/var/lib/docker:ro \
             -p 8080:8080 \
             --link=influxdb:influxdb \
             -d --name=cadvisor \
             google/cadvisor:latest \
             -storage_driver=influxdb \
             -storage_driver_host=influxdb:8086 \
             -storage_driver_db=db
```

在一个多主机配置中，你需要在所有节点上运行 cAdvisor 容器。InfluxDB 则以分布式方式部署在多台主机上，Grafana 则可能会在用于负载平衡的 Nginx 代理之后运行。

考虑到这些软件的开发节奏比较快，镜像也一直会变化，为了让整个系统可以正常工作，你可能需要对上面的 `docker run` 命令进行调整。

9.12 使用Weave Scope对容器布局进行可视化

9.12.1 问题

基于微服务架构（http://martinfowler.com/articles/microservices.html）构建的分布式应用程序会导致有数百（甚至更多）的容器在你的数据中心运行。对应用程序及组成该应用程序的所有容器进行可视化就显得很关键，也是整体基础设施的重要部分。

9.12.2 解决方案

来自 Weaveworks（http://weave.works）的 Weave Scope 提供了一个简单而强大的方式来对基础设施进行检测，并动态生成所有容器的架构图。它为你提供了多个维度：你可以按容器、镜像、主机或者应用程序对容器进行分组，并查看其特征详细信息。

Weave Scope 是开源的，你可以在 GitHub（https://github.com/weaveworks/scope）上查看它的源代码。

为了方便测试，像本书的其他范例一样，我也准备了一个 Vagrant 虚拟机。你可以使用 Git 克隆这个仓库，然后启动这个 Vagrant 虚拟机，如下所示。

```
$ git clone https://github.com/how2dock/docbook.git
$ cd how2dock/ch09/weavescope
$ vagrant up
```

这个 Vagrant 虚拟机会安装最新版本的 Docker（本书写作时的最新版本为 1.6.2）和 Docker Compose（参见范例 7.1 ）。在虚拟机的 /vagrant 文件夹下，你将会看到一个 docker-compose.yml 文件，这个文件定义了一个三层架构的应用程序，它包含两个负载平衡容器，两个应用程序容器和三个数据库容器。这是一个演示用的应用程序，用来对 Weave Scope 进行讲解。在虚拟机启动完成之后，通过 ssh 登录到虚拟机，移动到 /vagrant 文件夹下，通过 Docker Compose 启动应用程序，并运行 Weave Scope 脚本（即 scope），如下所示。

```
$ vagrant ssh
$ cd /vagrant
$ docker-compose up -d
$ ./scope launch
```

最后你将会有八个容器在运行：七个是演示用应用程序的容器，另一个是 Weave Scope。你可以通过 http://192.168.33.10:8001 或 http://192.168.33.10:8002 来访问测试用的应用程序。当然，最有趣的部分还是 Weave Scope 的仪表盘。打开浏览器访问 http://192.168.33.10:4040，你就会看到如图 9-5 所示的页面。

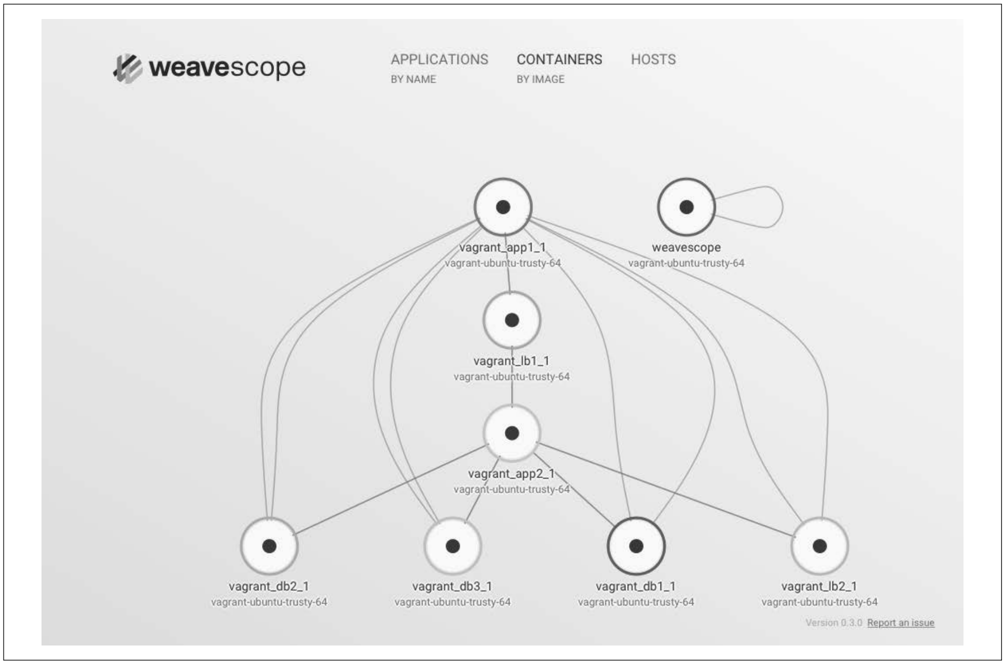


图 9-5：Weave Scope 仪表盘

通过页面导航，你可以浏览一下它不同的分组功能，查看一下各容器的详细信息。

9.12.3 讨论

Weave Scope 还处于早期开发阶段，你可以期待将来会有更多功能添加到这个开源项目。花些时间关注一下这个 Docker 容器可视化解决方案是绝对值得的。

从 Weave Scope 源代码进行构建也很简单，它提供了一个 Makefile 来构建 Docker 镜像。

9.12.4 参考

- 使用 Weave Scope 对 Docker 容器进行检测、映射和监视（http://thenewstack.io/how-to-detect-map-and-monitor-docker-containers-with-weave-scope-from-weaveworks/）

第10章

应用用例

10.0　简介

在本书的结尾，我想再次强调，使用 Docker 可以轻松地构建分布式应用程序。现在你的武器库里已经拥有了所有工具，使用这些工具，可以在你的数据中心内部或者外部构建可扩展的微服务应用。至少部署现有的分布式系统 / 框架变得更容易了，因为你只需要启动几个容器即可。Docker Hub（https://hub.docker.com）上汇集了诸如 MongoDB、ElasticSearch、Cassandra 以及其他很多镜像。假如你对镜像的内容感兴趣，下载这个镜像后启动一个或多个容器就可以了。

作为全书的最后一章，本章将会介绍几个被视为难题的用例，这些用例可以帮助你以自己的方式构建应用程序。首先，在范例 10.1 中，Pini Reznik 将会为你介绍如何使用 Docker 和 Jenkins 构建一个持续集成工作流。接着，他将会演示如何对这一工作流进行扩展，而在范例 10.2 中，他使用 Mesos 构建了一个持续部署工作流。

在范例 10.3 中，我们会介绍一个高级范例，向你演示如何构建一个动态负载平衡设置。这个设置利用了 `registrator` 和相应的基于 `consul` 和 `confd` 的键值存储。`confd` 是一个用于管理配置模板的系统。它监视键值存储中的键，并在键发生变化的时候自动基于模板重新生成配置文件。使用这种方案，举例来说，你可以在添加新的后端时自动重新配置负载平衡器服务。这也是构建一个弹性负载平衡器的关键所在。

在范例 10.4 中，我们将会构建一个兼容 S3 的对象存储系统，该系统由在 Kubernetes 之上运行的 Cassandra 和 pithos（http://pithos.io）构成。这个 pithos 软件提供了一套兼容 S3 的 API，并在 Cassandra 中管理存储桶。这个系统通过 Kubernetes 的 replication controller 实现了自动扩展。

在范例 10.5 和范例 10.6 中，我们将使用 Docker Network 构建一个 MySQL Galera 集群。Docker Network 在本书编写之际还处于实验阶段，但是本范例会帮助你深入理解 Docker Network 所带来的可能性。通过自动容器链接，MySQL Galera 集群中的节点可以在一个多主机网络中发现它们自己，并构建一个集群，就好像容器都是在同一台主机上运行一样。这非常强大，而且能简化分布式应用程序的设计。

作为本章的结尾，我们会以部署一个称为 Spark（http://spark.apache.org）的大规模数据处理系统为例。你可以在 Kubernetes 集群上运行 Spark，在基于 Docker Network 的基础设施中运行 Spark 也非常简单。最后的这个范例将会对如何通过 Docker Network 运行 Spark 进行介绍。

享受这最后一章，希望它能激发你的兴趣。

10.1　CI/CD：构建开发环境

——本范例由 Pini Reznik 提供

10.1.1　问题

你的 Node.js 应用开发环境需要具有一致性和可重复性。你不希望每次对 Node.js 源代码进行修改时都去重新构建 Docker 镜像。

10.1.2　解决方案

创建一个包括所有依赖的 Docker 镜像。在开发阶段挂载额外的卷，使用 Dockerfile 中的 ADD 指令向其他开发人员分发应用程序的镜像。

首先你需要一个 Node.js 的 Hello World 应用，这个应用由两个文件构成，如下所示。

app.js：

```
// 加载http模块来创建http服务器
var http = require('http');

// 配置我们的HTTP服务器,让它对所有请求返回Hello World
var server = http.createServer(function (request, response) {
  response.writeHead(200, {"Content-Type": "text/plain"});
  response.end("Hello World");
});

// 监听在8000端口,IP地址使用默认的"0.0.0.0"
server.listen(8000);

// 在终端上显示提示信息
console.log("Server running at http://127.0.0.1:8000/");
```

package.json：

```
{
```

```
  "name": "hello-world",
  "description": "hello world",
  "version": "0.0.1",
  "private": true,
  "dependencies": {
    "express": "3.x"
  },
  "scripts": {"start": "node app.js"}
}
```

你可以使用下面的 Dockerfile 文件来构建镜像。

```
FROM google/nodejs

WORKDIR /app
ADD package.json /app/
RUN npm install
ADD . /app

EXPOSE 8000
CMD []
ENTRYPOINT ["/nodejs/bin/npm", "start"]
```

这个 Dockerfile 安装了所有应用程序的依赖库，并将应用程序复制到了镜像中，你可以通过 `ENTRYPOINT` 指令启动该应用程序。

> Dockerfile 中指令的顺序很重要。之所以在其他操作之前添加 package.json 文件和安装依赖包，原因在于当你修改了应用程序代码，但是依赖包没有发生变化时，这样做会缩短镜像构建的时间。这是因为，在 ADD 指令所复制的文件中，任何文件的变化都会导致 Docker 构建缓存失效，后面的所有指令都要重复执行一遍。

准备好你的三个文件之后，你就可以构建镜像并启动容器了，如下所示。

```
$ docker build -t my_nodejs_image .
$ docker run -p 8000:8000 my_nodejs_image
```

这将会启动一个容器，容器中运行的是通过 ADD 指令添加到镜像中的应用程序。为了能够测试应用程序的变动，可以将本地源代码通过卷的方式挂载到容器中，命令如下所示。

```
$ docker run -p 8000:8000 -v "$PWD":/app my_nodejs_image
```

该命令会将最新代码所在的当前文件夹挂载到容器中的应用程序文件夹下。通过这种方式，你无需重新构建镜像就可以对最新的源代码进行调试。

为了与其他开发人员共享镜像以及将镜像推送到其他测试环境，你可以使用 Docker registry。下面的命令用来构建镜像并将镜像推送到指定的 Docker registry。

```
$ docker build -t <docker registry URL>:<docker registry port> \
/containersol/nodejs_app:<image tag>
$ docker push <docker registry URL>:<docker registry port>\
/containersol/nodejs_app:<image tag>
```

为了简化开发环境的工作，方便在将来集中为一个测试环境，可以使用以下三个脚本：build.sh、test.sh 和 push.sh。这些脚本将会变成单一的命令接口，你可以在开发过程中使用这些脚本完成所需的常见操作。

build.sh：

```
#!/bin/bash

# 传递给此脚本的第一个参数将用作镜像版本
# 如果未传递任何参数,则会将latest用作标签
if [ -z "${1}" ]; then
   version="latest"
else
   version="${1}"
fi

cd nodejs_app
docker build -t localhost:5000/containersol/nodejs_app:${version} .
cd ..
```

test.sh：

```
#!/bin/bash

# 传递给此脚本的第一个参数将用作镜像版本
# 如果未传递任何参数,则会将latest用作标签
if [ -z "${1}" ]; then
   version="latest"
else
   version="${1}"
fi
docker run -d --name node_app_test -p 8000:8000 -v "$PWD":/app localhost:5000/ \
containersol/nodejs_app:${version}

echo "Testing image: localhost:5000/containersol/nodejs_app:${version}"

# 允许Web服务器启动
sleep 1

# 如果下面URL的网页包含单词"success",
# 则表示测试成功
curl -s GET http://localhost:8000 | grep success
status=$?

# 清理测试用的容器
docker kill node_app_test
docker rm node_app_test

if [ $status -eq 0 ] ; then
   echo "Test succeeded"
else
   echo "Test failed"
fi

exit $status
```

push.sh：

```
#!/bin/bash

# 传递给此脚本的第一个参数将用作镜像版本
# 如果未传递任何参数,则会将latest用作标签
if [ -z "${1}" ]; then
   version="latest"
else
   version="${1}"
fi

docker push localhost:5000/containersol/nodejs_app:"${version}"
```

现在你就可以使用下面的命令来构建、测试和向 Docker registry 推送构建完成的镜像，如下所示。

```
$ ./build.sh <version>
$ ./test.sh <version>
$ ./push.sh <version>
```

10.1.3 讨论

保持一套在包括开发计算机在内的不同环境下都可以运行的、一致的用于构建、测试和部署的命令，通常来说都是一个很好的做法。这样，开发人员就可以像在持续集成环境下一样对应用程序进行测试，从而在早期发现与环境相关的问题。

这个例子使用了简单的脚本文件，但是更通用的做法是使用诸如 Maven 或者 Gradle 这样的构建系统来完成同样的任务。这两种构建系统都有 Docker 的插件，都可以使用类似编译和打包代码一样的行为，完成镜像构建和镜像推送工作。

我们目前的测试环境只有一个容器，但是，如果你需要一个多容器环境，可以使用 `docker-compose` 来创建多容器环境，同时使用更适合的测试系统（比如 Selenium）来替换简单的 `curl/grep`。你也可以在 Docker 容器中使用 Selenium，通过 `docker-compose` 将 Selenium 和应用的其他容器统一部署。

10.2 CI/CD：使用Jenkins和Apache Mesos构建持续交付工作流

——本范例由 Pini Reznik 提供

10.2.1 问题

你希望为由 Docker 容器组成的应用程序构建一个持续交付的工作流。

10.2.2 解决方案

设置一个 Jenkins 持续集成服务器，如果应用程序通过测试，则可以将应用程序部署到 Mesos 集群上。

图 10-1 是本范例中将会使用的环境的示意图。我们的目标是从开发环境将应用程序打包为 Docker 镜像，如果测试通过，就将镜像推送到 Docker registry，然后告诉 Marathon 在 Mesos 中对应用程序进行调度。

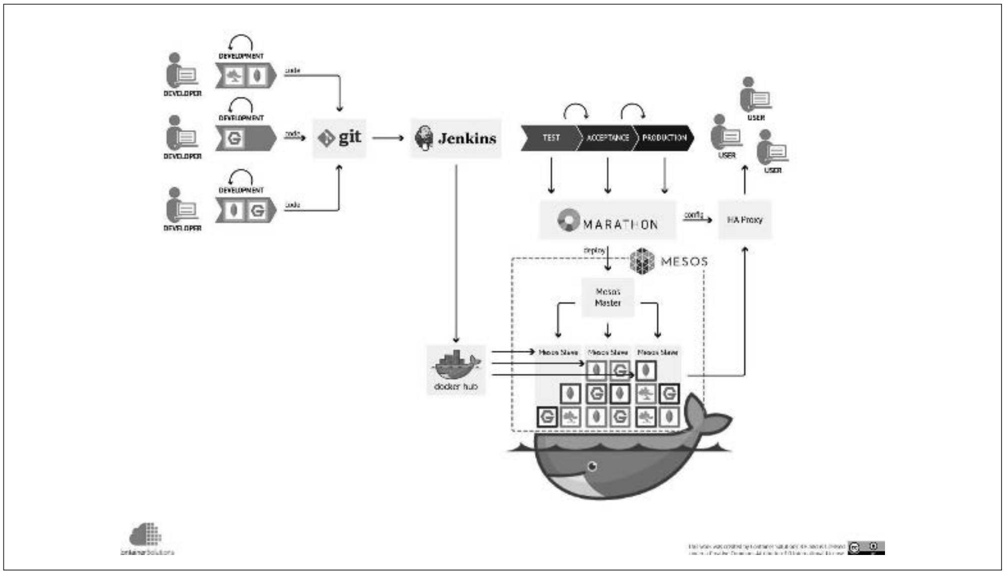


图 10-1：使用 Jenkins 和 Apache Mesos 的持续交付工作流

本范例使用了范例 10.1 中的例子。你也可以参考范例 7.2 来了解如何为开发构建一个 Mesos 集群。

首先你需要设置 Jenkins 服务器。最简单的方法是使用下面的 Docker Compose 配置文件。

```
jenkins:
  image: jenkins
  volumes:
  - jenkins-home:/var/jenkins_home
ports:
    - "8080:8080"
```

在上面的 Docker Compose 配置文件中定义的卷用于持久性存储，以避免构建配置信息和数据在 Jenkins 容器每次重新启动后丢失。而容器外部的文件夹则由其所有者负责备份和维护。

通过下面的命令启动 Docker Compose。

```
$ docker-compose up
```

这样，你就得到了一个可以工作的 Jenkins 服务器，该服务器在 http://localhost:8080 上运行。

这是一个很简单的任务，但不幸的是，它并没有什么用处。因为你还需要构建一个镜像，并将你的应用程序打包进去，然后从新构建的镜像启动容器来对应用程序进行测试。在标准 Docker 容器中不可能完成这些工作。

为了解决这个问题，你可以在 docker-compose.yml 文件中多添加两行，如下所示。

```
jenkins:
  image: jenkins
  volumes:
  - jenkins-home:/var/jenkins_home
  - /var/run/docker.sock:/var/run/docker.sock
  - /usr/bin/docker:/usr/bin/docker
  ports:
  - "8080:8080"
```

上面的代码增加了两个新的卷，一个挂载了 Docker 的 socket 文件，用于在 Docker 客户端和服务器之间进行通信，另一个卷挂载了 Docker 可执行程序作为 Docker 客户端。这样一来，你就可以在 Jenkins 容器中运行 Docker 命令了，这些命令会在宿主机和 Jenkins 容器中并行执行。

要想在一个功能完整的 Jenkins 服务器中运行 Docker 命令，另一个障碍是权限问题。默认情况下，只有 `root` 用户或者 `docker` 组内的用户才能访问 /var/run/docker.sock 文件。默认的 Jenkins 容器通过 `jenkins` 用户启动服务器。`jenkins` 用户不属于 `docker` 用户组，不过即使 `jenkins` 用户属于容器内的 `docker` 用户组，仍然不能访问 Docker socket，因为宿主机和容器中的组 ID 与用户 ID 并不一致（有一个例外就是 `root` 用户，其用户 ID 总是 0）。

要想解决这个问题，你需要使用 `root` 用户来启动 Jenkins 服务器。

为实现上述目的，你需要在 docker-compose.yml 文件中添加一条新的 `user` 指令，如下所示。

```
jenkins:
  image: Jenkins
  user: root
  volumes:
  - jenkins-home:/var/jenkins_home
  - /var/run/docker.sock:/var/run/docker.sock
  - /usr/bin/docker:/usr/bin/docker
  ports:
  - "8080:8080"
```

现在 Jenkins 服务器可以正常工作了，你可以开始部署范例 10.1 中讲过的 Node.js 应用了。

在 Node.js 的范例中，你已经有了用于构建、测试和向 Docker registry 推送镜像的脚本。现在你需要添加用于在 Mesos 中进行应用程序调度的配置文件，使用 Marathon 和另外的脚本来部署应用程序。

我们将用于在 Marathon 中部署应用程序的配置文件命名为 app_marathon.json，如下所示。

```
{
  "id": "app",
```

```
  "container": {
    "docker": {
      "image": "localhost:5000/containersol/nodejs_app:latest",
      "network": "BRIDGE",
      "portMappings": [
          {"containerPort": 8000, "servicePort": 8000}
      ]
    }
  },
  "cpus": 0.2,
  "mem": 512.0,
  "instances": 1
}
```

上面的配置使用了我们应用程序的 Docker 镜像，并且会使用 Jenkins 来构建这个镜像，然后通过 Marathon 来将这个镜像部署到 Mesos 上。这个文件还定义了应用程序所需要的资源，也可以包括健康检查的配置。

整个配置的最后一部分是部署脚本，这个脚本也会在 Jenkins 中执行。

deploy.sh：

```
#!/bin/bash

marathon=<Marathon URL>

if [ -z "${1}" ]; then
   version="latest"
else
   version="${1}"

fi

# 销毁旧的应用程序
curl -X DELETE -H "Content-Type: application/json" \
     http://${marathon}:8080/v2/apps/app

# 这时我们可以查询Marathon,直到应用程序停止
sleep 1

# 下面这些行将会创建一个app_marathon.json文件的副本,
# 然后更新镜像版本。由于在Marathon配置文件不支持使用
# 变量,所以我们必须这样来修改镜像的标签
cp -f app_marathon.json app_marathon.json.tmp
sed -i "s/latest/${version}/g" app_marathon.json.tmp

# 将应用提交到Marathon
curl -X POST -H "Content-Type: application/json" \
     http://${marathon}:8080/v2/apps \
     -d@app_marathon.json.tmp
```

现在你就可以通过 `docker-compose` 命令来启动 Jenkins 服务器了，然后在 Jenkins 任务配置中定义执行步骤。图 10-2 显示了如何进行设置。

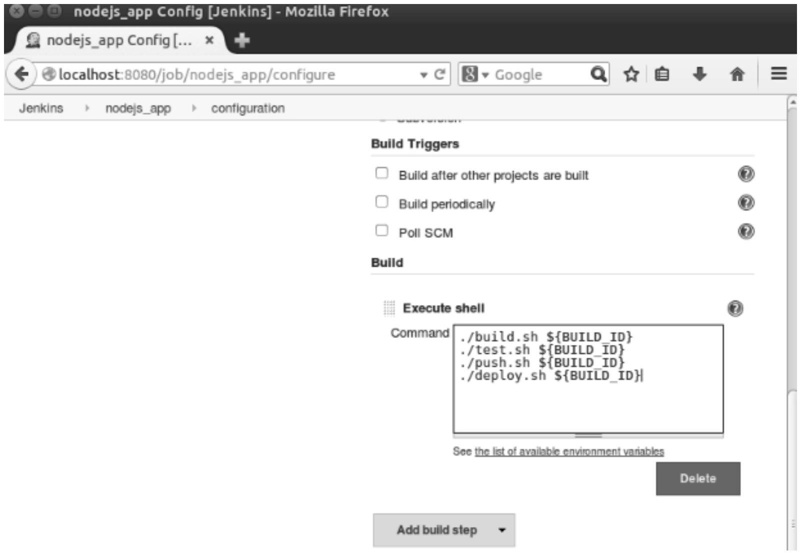


图 10-2：Jenkins 任务界面

10.2.3　讨论

有多种方式可以用来解决在容器内启动容器的问题。其中一种方式是，将 Docker socket 挂载到容器中来实现 Docker 服务器与客户端之间的通信。其他的方法可能包括直接在以特权模式启动的容器中启动容器。另一种方式是将 Docker 服务器配置为接收远程 API 调用，并将 Jenkins 容器内的 Docker 客户端配置为使用一个完整的 URL 来与 Docker 服务器进行通信。这需要配置网络以允许在服务器和客户端之间进行通信。

10.3　ELB：使用confd和registrator创建动态负载平衡器

10.3.1　问题

你想建立一个动态负载平衡器，当容器启动或者停止时，能够动态更新配置。

10.3.2　解决方案

我们的解决方案会使用 `registrator`（用于进行服务发现，参见范例 7.13）和 `confd`（https://github.com/kelseyhightower/confd），后者可以用来从键值存储中获取 `registrator` 所需要的信息，并根据模板更新配置文件。

为了演示这一方案，你需要构建一个简单的单节点部署环境。这是一个简单的 hostname 应

用程序，但是它会在多个容器中运行。在这个应用程序的容器之前，会放置一个 Nginx 负载平衡器，用于将访问请求分发到应用程序的容器上。这些应用程序的容器会自动注册到 Consul 键值存储，这由 `registrator` 来实现。而 `confd` 则会从 Consul 拉取信息，然后写入 Nginx 的配置文件。最后负载平衡器（即 Nginx）将会使用新的配置文件重启。图 10-3 是这个例子的示意图。

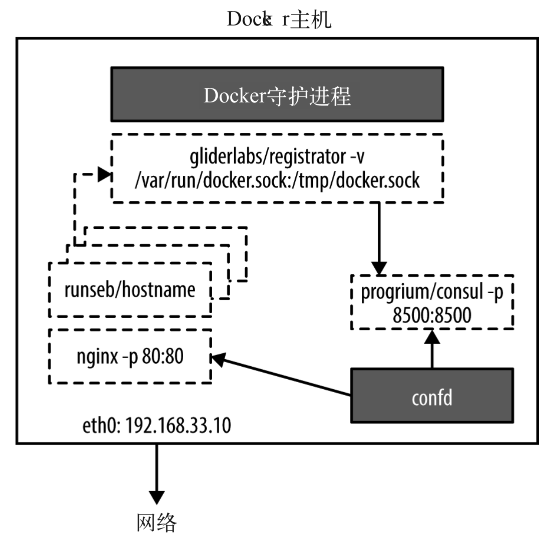


图 10-3：动态负载平衡原理

作为开始，你需要重复在范例 7.3 中介绍的步骤。你需要在一个容器中运行基于 Consul 的键值存储。

> 在生产环境中，你可能会使用多节点键值存储，并与应用程序运行在不同的节点之上。

你可以通过 progrium/consul 镜像来完成这上述功能，如下所示。

```
$ docker run -d -p 8400:8400 -p 8500:8500 -p 8600:53/udp
             -h cookbook progrium/consul -server
             -bootstrap -ui-dir /ui
```

然后需要启动 registrator 容器，并将 registry 的 URI 设置为 consul://192.168.33.10:8500/elb。实际上，你的 Docker 主机的 IP 地址应该与下面的例子不同。

```
$ docker run -d -v /var/run/docker.sock:/tmp/docker.sock
             -h 192.168.33.10 gliderlabs/registrator
             -ip 192.168.33.10 consul://192.168.33.10:8500/elb
```

接着，你需要启动应用程序。拉取 `runseb/hostname` 镜像（这是一个简单的应用程序），将会返回该容器的 ID。我们先启动两个这样的容器，如下所示。

```
$ docker run -d -p 5001:5000 runseb/hostname
$ docker run -d -p 5002:5000 runseb/hostname
```

如果检查一下 Consul 的界面，你会发现这两个容器已经正确注册了，这要归功于 `registrator`，如图 10-4 所示。

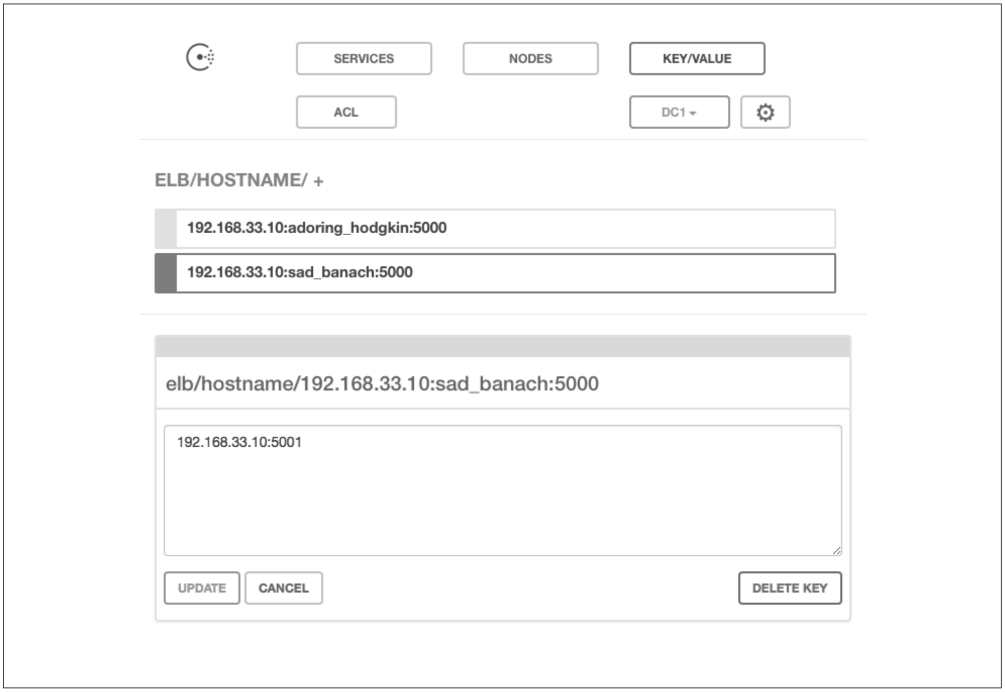


图 10-4：Consul 动态负载平衡节点

创建一个 Nginx 配置文件，作为这两个应用程序容器的负载平衡器。假设你的 Docker 宿主机的 IP 地址是 `192.168.33.10`，则可以使用下面这个例子。

```
events {
    worker_connections  1024;
}

http {
    upstream elb {
      server 192.168.33.10:5001;
      server 192.168.33.10:5002;
    }

    server {
        listen 80;
        location / {
```

```
                proxy_pass http://elb;
            }
        }
    }
```

接着，启动 Nginx 容器，将容器的 80 端口绑定到宿主机的 80 端口上，并将配置文件挂载到容器中。同时，为这个容器设置一个名称以方便后续操作，如下所示。

```
$ docker run -d -p 80:80 -v /home/vagrant/nginx.conf:/etc/nginx/nginx.conf
             --name elb nginx
```

到这一步，你已经设置了一个基本负载平衡器。Nginx 容器向宿主机公开了 80 端口，并对两个应用程序的容器进行负载平衡。如果你使用 `curl` 向 Nginx 容器发起 HTTP 请求，将会得到两个应用容器的容器 ID。结果看起来如下所示。

```
$ curl http://192.168.33.10
8eaab9c31e1a
$ curl http://192.168.33.10
a970ec6274ca
$ curl http://192.168.33.10
8eaab9c31e1a
$ curl http://192.168.33.10
a970ec6274ca
```

到这里，除了容器注册之外，其他都还不是自动完成的。为了能够在容器启动或者停止时对 Nginx 重新进行配置，你需要一种机制来监视 Consul 的键，并在其值发生变更时，更新 Nginx 的配置文件。这时候就轮到 `confd`（https://github.com/kelseyhightower/confd）大展身手了。你可以从 `confd` 在 GitHub 上的页面（https://github.com/kelseyhightower/confd/releases）下载 confd 的二进制文件。

它的快速入门指南（https://github.com/kelseyhightower/confd/blob/master/docs/quick-start-guide.md）非常不错。但是我们只会涉及一些基本步骤。首先，让我们创建一个用于保存配置文件模板的文件夹，如下所示。

```
sudo mkdir -p /etc/confd/{conf.d,templates}
```

接着来创建一个资源模板 resource config。这个文件用来告诉 `confd` 你想要管理的配置模板的位置，以及当所监控的值发生变化时，将配置文件写入到哪里。在 /etc/confd/conf.d/config.toml 文件中写入如下内容。

```
[template]
src = "config.conf.tmpl"
dest = "/home/vagrant/nginx.conf"
keys = [
    "/elb/hostname",
]
```

现在让我们编辑 Nginx 模板文件 /etc/confd/templates/config.conf.tmpl。这些模板都是 Golang 规则的模板（http://golang.org/pkg/text/template/#pkg-overview），因此任何在 Golang 模板中可以使用的语法在这个模板中都可以使用。

```
events {
    worker_connections  1024;
}

http {
    upstream elb {
     {{range getvs "/elb/hostname/*"}}
      server {{.}};
     {{end}}
    }

    server {
        listen 80;

        location / {
            proxy_pass http://elb;
        }
    }
}
```

这个模板是一个最简单的 Nginx 负载平衡配置文件。你会看到定义为 elb 的上游将包含将从 Consul 中存储的键 /elb/hostname/ 中提取的一组服务器。

现在你的模板已经准备就绪，让我们试一下 confd 的单次模式。也就是说，你需要手动调用 confd，并指定需要使用的后端服务（在这个例子中为 Consul），然后 confd 将会写入文件 /home/vagrant/nginx.conf（通过 config.toml 文件中的 dest 项定义该文件位置），如下所示。

```
$ ./confd -onetime -backend consul -node 192.168.33.10:8500
```

由于在前面启动 Nginx 容器时你已经手动编辑了 nginx.conf 文件，因此上面 confd 生成的配置文件内容将与手动编辑的文件一模一样。现在就让我们再启动几个新容器，并再次运行 confd 命令，如下所示。

```
$ docker run -d -p 5003:5000 runseb/hostname
$ ./confd -onetime -backend consul -node 192.168.33.10:8500
... ./confd[832]: WARNING Skipping confd config file.
... ./confd[832]: INFO /home/vagrant/nginx.conf has md5sum \
acf6552d92cb9eb79b1068cf40b8ec0f should be 001894b713827404d0c5e72e2a66844d
... ./confd[832]: INFO Target config /home/vagrant/nginx.conf out of sync
... ./confd[832]: INFO Target config /home/vagrant/nginx.conf has been updated
```

你会看到 confd 检测到配置已经发生变化，并重新生成了新的配置文件。当我们启动新的应用程序容器时，registrator 会自动将其注册到 consul，confd 就能检测到配置的变化并重新生成配置文件。由于你通过一次性命令来重新生成了配置文件，现在让我们重新启动 Nginx 容器，你将看到 Nginx 容器会使用新的配置（可以通过挂载到 Nginx 容器的卷来访问），如下所示。

```
$ docker restart elb
$ curl http://192.168.33.10
a970ec6274ca
$ curl http://192.168.33.10
```

```
8eaab9c31e1a
$ curl http://192.168.33.10
71d8297c1538
```

剩下的唯一要做的就是让 `confd` 在守护进程模式下运行，并且在配置发生变化时，重启 Nginx 容器。这可以通过编辑 /etc/confd/conf.d/config.toml 文件，添加一条 `reload_cmd` 指令来重启 Nginx，如下所示（假设你的 Nginx 容器已经像前面所示命名为 elb）。

```
[template]
src = "config.conf.tmpl"
dest = "/home/vagrant/nginx.conf"
keys = [
    "/elb/hostname",
]
reload_cmd = "docker restart elb"
```

最后，在守护进程模式下启动 `confd`。为了方便测试，可以设置一个很小的从 Consul 拉取信息的间隔。然后就可以随意启动或者停止应用程序的容器了。你将会看到，每次你启动或者停止、终止一个应用程序容器时，`confd` 都会动态更新你的配置文件并重启 elb 容器，如下所示。

```
$ ./confd -backend consul -interval 5 -node 192.168.33.10:8500
... ./confd[1463]: WARNING Skipping confd config file.
... ./confd[1463]: INFO /home/vagrant/nginx.conf has md5sum \
acf6552d92cb9eb79b1068cf40b8ec0f should be 001894b713827404d0c5e72e2a66844d
... ./confd[1463]: INFO Target config /home/vagrant/nginx.conf out of sync
... ./confd[1463]: INFO Target config /home/vagrant/nginx.conf has been updated
... ./confd[1463]: INFO /home/vagrant/nginx.conf has md5sum \
001894b713827404d0c5e72e2a66844d should be cecb5ddc469ba3ef17f9861cde9d529a
... ./confd[1463]: INFO Target config /home/vagrant/nginx.conf out of sync
... ./confd[1463]: INFO Target config /home/vagrant/nginx.conf has been updated
... ./confd[1463]: INFO /home/vagrant/nginx.conf has md5sum \
cecb5ddc469ba3ef17f9861cde9d529a should be 0b97f157f437083ffba43f93a426d28f
... ./confd[1463]: INFO Target config /home/vagrant/nginx.conf out of sync
... ./confd[1463]: INFO Target config /home/vagrant/nginx.conf has been updated
```

这就是基于 Docker 的动态负载平衡。为了让它更具弹性，你需要监视这些容器的负载，并自动创建新的容器，新容器的启动也会触发 elb 配置文件的重新生成。

10.3.3　讨论

本范例非常长且包含很多步骤。为了方便测试，我像其他章节一样，准备了一个 Vagrant 镜像，命令如下所示。

```
$ git clone https://github.com/how2dock/docbook.git
$ cd docbook/ch10/confd
$ vagrant up
$ vagrant ssh
```

这个虚拟机启动后会下载所有镜像，你可以直接使用，如下所示。

```
$ docker images
REPOSITORY              TAG       IMAGE ID        CREATED         VIRTUAL SIZE
```

```
progrium/consul          latest    e66fb6787628    10 days ago     69.43 MB
nginx                    latest    319d2015d149    3 weeks ago     132.8 MB
runseb/hostname          latest    7c9d1ddd2ceb    3 months ago    349.3 MB
gliderlabs/registrator   latest    b1c29d1a74a9    4 months ago    11.79 MB
```

并且 `confd` 配置文件也已经保存到了 /etc/confd/conf.d/config.toml 和 /etc/confd/templates/config.conf.tmp。

你只需要启动这些容器即可，如下所示。

```
$ docker run -d -p 8400:8400 -p 8500:8500 -p 8600:53/udp
             -h cookbook progrium/consul -server
             -bootstrap -ui-dir /ui
$ docker run -d -v /var/run/docker.sock:/tmp/docker.sock
             -h 192.168.33.10 gliderlabs/registrator
             -ip 192.168.33.10 consul://192.168.33.10:8500/elb
$ docker run -d -p 80:80 -v /home/vagrant/nginx.conf:/etc/nginx/nginx.conf
             --name elb nginx
$ docker run -d -p 5001:5000 runseb/hostname
$ docker run -d -p 5002:5000 runseb/hostname
```

你可以通过 Docker Compose 来启动所有容器（参见范例 7.1）。

然后运行 `confd`，如下所示。

```
$ ./confd -backend consul -interval 5 -node 192.168.33.10:8500
```

10.3.4 参考

- confd 快速入门（https://github.com/kelseyhightower/confd/blob/master/docs/quick-start-guide.md）

10.4 DATA：使用Cassandra和Kubernetes构建兼容S3的对象存储

10.4.1 问题

你希望构建自己的兼容 S3（http://aws.amazon.com/s3/）的对象存储。

10.4.2 解决方案

Amazon S3 是领先的基于云的对象存储服务。由于 S3 已经广泛使用，一些存储后端已经开发了兼容 S3 的 API 并将它们放到存储系统前端：RiakCS（http://docs.basho.com/riakcs/latest/）、GlusterFS（http://www.gluster.org）和 Ceph（http://ceph.com）等。分布式数据库

Apache Cassandra（http://cassandra.apache.org）也是一个不错的选择，最近一个名为 Pithos（http://pithos.io）的项目已经开始基于 Cassandra 构建一个兼容 S3 的对象存储软件。

这特别有趣，因为 Cassandra 在企业级被广泛使用。但是对 Docker 来说，这可能比较具有挑战性，因为你需要使用 Docker 容器构建一个 Cassandra 集群。值得庆幸的是，有了 Kubernetes 这样的集群管理工具和容器编排系统，运行一个基于 Docker 的 Cassandra 集群已经比较简单了。Kubernetes 官方文档也提供了一个关于如何运行 Cassandra 集群的例子（https://github.com/GoogleCloudPlatform/kubernetes/tree/master/examples/cassandra）。

因此，为了构建我们自己的 S3 对象存储，你需要在 Kubernetes 上运行一个 Cassandra 集群，并将 Pithos 作为前端来提供兼容 S3 的 API。

你也可以使用 Docker Swarm 来完成同样的工作。

在开始之前，你需要先有一个 Kubernetes 集群。最简单的方式是使用 Google 容器引擎（参见范例 8.10）。如果你不想使用 Google 容器引擎或者需要学习一下 Kubernetes，可以参考一下第 5 章，你将会学习到如何部署自己的 Kubernetes 集群。不管采用哪种技术，在继续之前，你需要能使用 `kubectl` 客户端来列出集群中已有的节点，如下所示。

```
$ ./kubectl get nodes
NAME                              LABELS                                    STATUS
k8s-cookbook-935a6530-node-hsdb   kubernetes.io/hostname=...-node-hsdb   Ready
k8s-cookbook-935a6530-node-mukh   kubernetes.io/hostname=...-node-mukh   Ready
k8s-cookbook-935a6530-node-t9p8   kubernetes.io/hostname=...-node-t9p8   Ready
k8s-cookbook-935a6530-node-ugp4   kubernetes.io/hostname=...-node-ugp4   Ready
```

现在一切准备就绪，可以开始构建 Cassandra 集群了。你可以直接使用 Kubernetes 自带的例子（https://github.com/GoogleCloudPlatform/kubernetes/tree/master/examples/cassandra），或者克隆我的仓库，如下所示。

```
$ git clone https://github.com/how2dock/docbook.git
$ cd docbook/ch05/examples
```

由于 Kubernetes 是一个快速演进的软件，它的 API 变化得也很快。pod、replication controller 和 service 规范文件可能需要根据最新的 API 版本进行调整。

然后启动 Cassandra replication controller，增加副本数量，并启动服务，如下所示。

```
$ kubectl create -f ./cassandra/cassandra-controller.yaml
$ kubectl scale --replicas=4 rc cassandra
$ kubectl create -f ./cassandra/cassandra-service.yaml
```

当镜像下载完成之后，你的 Kubernetes pod 将会进入运行状态。需要注意的是，当前使用的镜像来自 Google 的 registry。这是由于这个镜像包含了 Cassandra 配置中指定的发现类。你也可以使用 Docker Hub 上的 Cassandra 镜像，但是必须要将这个 Java 类放到 Cassandra 镜像中，让 Cassandra 节点能彼此互相发现。修改副本数量可以让你对 Cassandra 集群进行扩展，启动一个服务可以让你为 Cassandra 集群公开一个 DNS 端点。

确认指定数量的 pod 是否正在运行，如下所示。

```
$ kubectl get pods --selector="name=cassandra"
```

当 Cassandra 发现所有节点并进行数据库存储的再平衡之后，你将会看到类似下面的结果。（这取决于你所设置的副本数量，并且 HOST ID 也可能不一样。）

```
$ ./kubectl exec cassandra-5f709 -c cassandra nodetool status
Datacenter: datacenter1
=======================
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address    Load       Tokens  Owns (effective) Host ID               Rack
UN  10.16.2.4  84.32 KB   256     46.0%            8a0c8663-074f-4987... rack1
UN  10.16.1.3  67.81 KB   256     53.7%            784c8f4d-7722-4d16... rack1
UN  10.16.0.3  51.37 KB   256     49.7%            2f551b3e-9314-4f12... rack1
UN  10.16.3.3  65.67 KB   256     50.6%            a746b8b3-984f-4b1e... rack1
```

你可以使用 `kubectl logs` 命令来方便地查看 pod 中容器的日志。

你已经拥有了一个可以正常工作的 Cassandra 集群，现在可以准备启动 Pithos 了，Pithos 提供了一个兼容 S3 的 API，并且使用 Cassandra 作为对象存储。

Pithos 是一个守护程序，为 Cassandra 集群提供一个兼容 S3 的前端。因此，如果你在 Kubernetes 集群中运行一个 Pithos，并且指向了同在 Kubernetes 集群中运行的 Cassandra 集群，那么就可以提供一套兼容 S3 的接口。

为此，我在 Docker Hub 上创建了一个 Pithos 的 Docker 镜像 runseb/pithos。这是一个自动构建的 Docker 镜像，因此你可以看到该镜像的 Dockerfile。这个镜像包括一个默认的配置文件。你需要修改这个配置文件中的访问密钥和存储桶存储定义。

现在你可以通过 Kubernetes replication controller 来启动 Pithos 了，并可以通过在 GCE 上创建的外部负载平衡器来公开服务。Pithos 可以通过 DNS 找到之前启动的 Cassandra 服务。

但是，你还需要设置对象存储的数据库模式。这是通过一个引导过程完成的。要完成此操作，你需要运行一个不会重启的 pod 来在 Cassandra 中安装 Pithos 数据库模式。使用之前克隆的 example 目录下的 YAML 文件，如下所示。

```
$ kubectl create -f ./pithos/pithos-bootstrap.yaml
```

等待引导过程的完成（即 pod 变为 succeed 状态）。然后启动 replication controller。到目前为止，你将只启动一个副本。使用 replication controller 可以轻松地挂载一个服务并通过一个公网 IP 地址来对外公开这个服务。

```
$ kubectl create -f ./pithos/pithos-rc.yaml
$ kubectl create -f ./pithos/spithos.yaml
$ ./kubectl get services --selector="name=pithos"
NAME      LABELS        SELECTOR      IP(S)            PORT(S)
pithos    name=pithos   name=pithos   10.19.251.29     8080/TCP
                                      104.197.27.250
```

由于 Pithos 默认会绑定到 8080 端口，所以请确保你在防火墙上打开了负载平衡器的公网 IP 地址。

当 Pithos 进入运行状态，你就得到了一个基于 Cassandra 的兼容 S3 的对象存储服务，并且在由 Kubernetes 负责管理的 Docker 容器中运行。恭喜你！

10.4.3 讨论

这个例子非常有趣，但是你需要能够使用它，并确认它是真的兼容 S3。为此，你可以使用一些常见的 S3 实用工具，比如 s3cmd 或 boto。

比如，你可以安装 s3cmd（http://s3tools.org/s3cmd），并创建一个如下所示的配置文件。

```
$ cat ~/.s3cfg
[default]
access_key = AKIAIOSFODNN7EXAMPLE
secret_key = wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
check_ssl_certificate = False
enable_multipart = True
encoding = UTF-8
encrypt = False
host_base = s3.example.com
host_bucket = %(bucket)s.s3.example.com
proxy_host = 104.197.27.250
proxy_port = 8080
server_side_encryption = True
signature_v2 = True
use_https = False
verbosity = WARNING
```

将上面的 `proxy_host` 替换为你的 Pithos 服务外部负载平衡器的实际 IP 地址。



这个例子使用了一个没有加密的代理。而且访问密钥也是存储在 Dockerfile（https://github.com/runseb/pithos）中的默认值，你需要修改这些设置。

有了上面的配置文件，现在就可以使用 `s3cmd` 来创建用于存储数据的存储桶了，如下所示。

```
$ s3cmd mb s3://foobar
Bucket 's3://foobar/' created
```

```
$ s3cmd ls
2015-06-09 11:19  s3://foobar
```

如果你想在 Python 中使用 Boto（https://github.com/boto/boto3），下面的代码也能正常工作。

```
#!/usr/bin/env python

from boto.s3.key import Key
from boto.s3.connection import S3Connection
from boto.s3.connection import OrdinaryCallingFormat

apikey='AKIAIOSFODNN7EXAMPLE'
secretkey='wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY'

cf=OrdinaryCallingFormat()

conn=S3Connection(aws_access_key_id=apikey,
                  aws_secret_access_key=secretkey,
                  is_secure=False,host='104.197.27.250',
                  port=8080,
                  calling_format=cf)

conn.create_bucket('foobar')
```

这就是本范例的所有内容。这些步骤听起来可能很多，但在有 Docker 之前，你不可能这么轻松就能运行一个 S3 对象存储服务。是 Docker 真正让运行分布式应用变得轻而易举。

10.5 DATA：使用Docker Network构建MySQL Galera集群

10.5.1 问题

你想使用新的 Docker Network（参见范例 3.14）功能构建一个基于两台 Docker 主机的 MySQL Galera 集群。Galera 是一个多主节点的高可用 MySQL 数据库解决方案。

10.5.2 解决方案

我们已经在范例 3.14 中介绍过了 Docker Network，可以使用 Docker Network 基于 VXLAN 协议来构建跨越多台 Docker 主机的覆盖网络。覆盖网络非常适合用来为容器分配在同一个可路由子网内的 IP 地址，以及通过更新每个容器中的 /etc/hosts 文件来进行名称解析。因此在同一个覆盖网络中运行的所有容器都可以通过容器名来互相访问。

这大大地简化了跨主机的网络问题，同时也使得很多已有的基于单台主机的方案在多主机的环境下也能继续使用。

在本书编写之际，Docker Network 还只能在实验预发布的 Docker 二进制程序（即 1.8.0-dev）中使用。在 1.9 中 Docker Network 应该就可以使用了。将来也可能不再需要使用 Consul 服务。

为了在两台 Docker 主机上构建 Galera（http://galeracluster.com）集群，你需要在每台主机上安装实验版的 Docker 二进制程序。然后你需要按照博客文章 http://galeracluster.com/2015/05/getting-started-galera-with-docker-part-1/ 里的步骤进行操作。本范例采用了与范例 3.14 相同的配置。你需要在每台 Docker 主机上启动多个来自 Docker Hub 的 erkules/galera:basic 容器。

按照惯例，让我们从本书附带仓库中的 Vagrant 虚拟机开始，如下所示。

```
$ git clone https://github.com/how2dock/docbook.git
$ cd docbook/ch10/mysqlgalera
$ vagrant up
$ vagrant status
Current machine states:

consul-server             running (virtualbox)
mysql-1                   running (virtualbox)
mysql-2                   running (virtualbox)
```

主机 `consul-server` 是现在 Docker Network 所必需的，但是将来可能会发生变化。现在我们将这台 Consul 服务器用作一个键值存储，每台主机上的 Docker 引擎使用 Consul 来存储各主机的信息。作为提醒，请检查一下 Vagrantfile 文件中启动时所设置的 `DOCKER_OPTS` 选项，你会看到我们也定义了一个名为 `multihost` 的默认覆盖网络。

当所有虚拟机都启动完成之后，请通过 `ssh` 到第一台主机使用 erkules/galera:basic 镜像启动 Galera 集群中的第一个节点。可以通过官方的参考（http://galeracluster.com/2015/05/getting-started-galera-with-docker-part-1/）来了解一下用于构建该镜像的 Dockerfile 文件内容。

让我们来启动 Galera 集群，如下所示。

```
$ vagrant ssh mysql-1
$ docker run -d --name node1 -h node1 erkules/galera:basic \
             --wsrep-cluster-name=local-test \
       --wsrep-cluster-address=gcomm://
```

再到 `mysql-2` 主机中启动另外两个 Galera 节点。请注意，你使用节点名 `node1` 作为集群的地址。这之所以会正常工作，是因为 Docker Network 会自动更新 /etc/hosts 中的记录，该文件会包括 `node1`、`node2` 和 `node3` 节点的 IP 地址。由于这三个容器都在同一个覆盖网络中，它们可以直接互相通信，而不需要通过端口映射、容器链接或者更复杂的网络设置，如下所示。

```
$ vagrant ssh mysql-2
$ docker run --detach=true --name node2 -h node2 erkules/galera:basic \
             --wsrep-cluster-name=local-test \
       --wsrep-cluster-address=gcomm://node1
```

```
$ docker run --detach=true --name node3 -h node3 erkules/galera:basic \
             --wsrep-cluster-name=local-test \
       --wsrep-cluster-address=gcomm://node1
```

现在回到虚拟机 mysql-1，你会看到经过很短的一段时间之后，在 mysql-2 上启动的两个集群节点已经加入到集群中，如下所示。

```
$ docker exec -ti node1 mysql -e 'show status like "wsrep_cluster_size"'
+--------------------+-------+
| Variable_name      | Value |
+--------------------+-------+
| wsrep_cluster_size | 3     |
+--------------------+-------+
```

确实，在 node1 容器的 /etc/hosts 文件中，已经增加了另外两个集群节点的 IP 地址记录。

```
$ docker exec -ti node1 cat /etc/hosts
...
172.21.0.6  node1.multihost
172.21.0.6  node1
172.21.0.8  node2
172.21.0.8  node2.multihost
172.21.0.9  node3
172.21.0.9  node3.multihost
```

本范例很有趣，因为通过使用 Docker Network，你可以在多台 Docker 主机之间使用与单台 Docker 主机完全相同的部署方法。

10.5.3 讨论

尝试添加更多的 Galera 节点，然后终止其中的一些节点，你就会看到集群的大小也会随之变化。

10.5.4 参考

- 在单台 Docker 主机上构建 Galera 集群（http://galeracluster.com/2015/05/getting-started-galera-with-docker-part-1/）
- 在多台 Docker 主机上构建 Galera 集群（http://galeracluster.com/2015/05/getting-started-galera-with-docker-part-2-2/）

10.6 DATA：以动态方式为MySQL Galera集群配置负载平衡器

10.6.1 问题

范例 10.5 充分利用了 Docker Network 功能创建了覆盖网络，在两台 Docker 主机上构建了一个多节点的 Galera 集群。现在你希望对负载平衡器进行自动配置，将负载分发到 Galera 集群的不同节点。

10.6.2 解决方案

使用与范例 10.3 相同的配置。使用 `registrator` 将 MySQL 节点以动态方式注册到键值存储，比如 Consul，使用 `confd` 来管理一个 `nginx` 模板，将负载分散到 Galera 集群节点上。图 10-5 描述了一个使用 Docker Network 的两节点方案，`registrator` 在每个节点上运行，用于发布服务，Nginx 在其中一个节点上运行，用于在这些节点之间进行负载平衡。

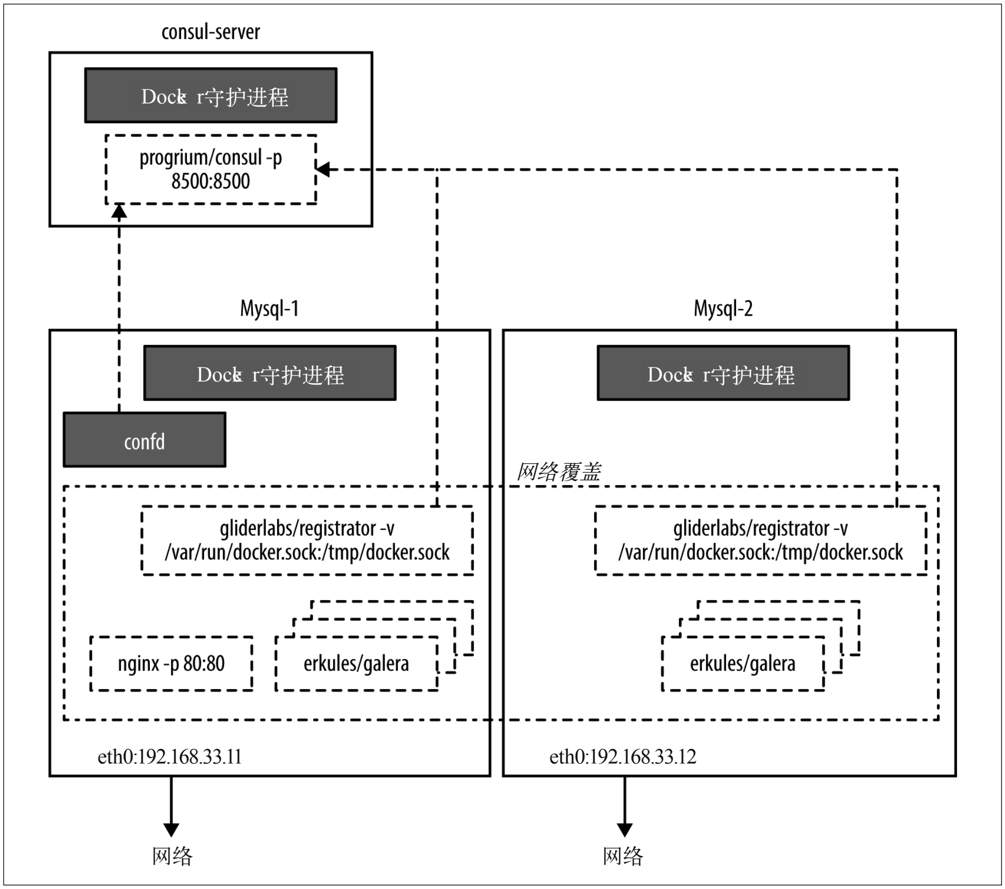


图 10-5：Galera 集群的动态负载平衡

在每个节点上都运行 `registrator`，并指向在 `192.168.33.10` 这台独立 VM 之上运行的 Consul 服务地址，使用 erkules/galera:basic 镜像启动 Galera 集群中最初的两个节点。

在 IP 地址为 `192.168.33.11` 的 `mysql-1` 主机上执行下面的命令。

```
$ docker run -d -v /var/run/docker.sock:/tmp/docker.sock
             gliderlabs/registrator
             -ip 192.168.33.11 consul://192.168.33.10:8500/galera
```

```
$ docker run -d --name node1
             -h node1 erkules/galera:basic
             --wsrep-cluster-name=local-test --wsrep-cluster-address=gcomm://
```

在 IP 地址为 192.168.33.12 的 mysql-2 主机上执行下面的命令。

```
$ docker run -d -v /var/run/docker.sock:/tmp/docker.sock
             gliderlabs/registrator
             -ip 192.168.33.12 consul://192.168.33.10:8500/galera
$ docker run -d --name node2
             -h node2 erkules/galera:basic
             --wsrep-cluster-name=local-test --wsrep-cluster-address=gcomm://node1
```

创建一个 Nginx 的配置文件，作为这两个应用程序的负载平衡器。假设你想让这个负载平衡器在 IP 地址 192.168.33.11 上运行，可以使用下面的例子。

```
events {
    worker_connections  1024;
}

http {
    upstream galera {
      server 192.168.33.11:3306;
      server 192.168.33.12:3306;
    }

    server {
        listen 80;
        location / {
            proxy_pass http://galera;
        }
    }
}
```

接下来启动你的 Nginx 容器，将容器的 80 端口绑定到宿主机的 80 端口上，并将配置文件挂载到容器中。同时，为容器设置一个名称，这会方便以后对该容器进行管理，如下所示。

```
$ docker run -d -p 3306:3306 -v /home/vagrant/nginx.conf:/etc/nginx/nginx.conf
             --name galera nginx
```

确认一下你的负载平衡器是否能正常工作。你可以回顾一下范例 10.3，使用该范例中介绍的方法进行验证。当你添加新的 MySQL 容器时，就会使用 confd 自动重新配置 Nginx 配置文件模板。

10.7　DATA：构建Spark集群

10.7.1　问题

你正在寻找一个能够并行工作、快速计算并访问大型数据集的数据处理引擎。你选择了 Apache Spark（http://spark.apache.org），并想以容器方式部署 Spark。

10.7.2 解决方案

Apache Spark（http://spark.apache.org）是一个非常快的数据处理引擎，支持大规模（有很多工作节点）集群，能对海量数据进行处理。Spark 支持 Java、Scala、Python 和 R 语言编程接口，是一个通过编程解决复杂数据处理问题的伟大工具。

Spark 集群可以在 Kubernetes（https://github.com/GoogleCloudPlatform/kubernetes/tree/master/examples/spark）集群上进行部署，不过随着 Docker Network 的发展，Kubernetes 上的部署流程也可以用于 Docker Network 之上。实际上，Docker Network（参见范例 3.14）会构建一个隔离的、跨越多台 Docker 主机的网络，管理简单的名称解析，并对外公开服务。

因此，要部署一个 Spark 集群，你需要使用 Docker Network 并进行如下操作。

- 使用 Google registry 的镜像启动一个 Spark 主节点，Google registry 在 Kubernetes 例子中已经见过了。
- 启动一组 Spark 工作节点，这些节点的镜像基于 Google registry 的镜像（https://registry.hub.docker.com/u/runseb/spark-worker/），并做了简单修改。

Spark 工作镜像的启动脚本将 Spark 主节点的端口硬编码为 `7077`，而并没有使用 Kubernetes 设置的环境变量。这个镜像可以在 Docker Hub（https://registry.hub.docker.com/u/runseb/spark-worker/）上找到，你可以参考一下 GitHub（https://github.com/runseb/spark-docker）上的启动脚本。

让我们先启动主节点，请确保你已经将主机名设置为 `spark-master`，如下所示。

```
$ docker run -d -p 8080:8080 --name spark-master -h spark-master gcr.io/ \
google_containers/spark-master
```

接着让我们来创建三个 Spark 工作节点。实际上，你也可以创建更多的工作节点，不过需要确保这些节点都位于同一个 Docker Network 之中，如下所示。

> 为了防止你的节点和（或）容器崩溃，需要限制一下为每个 Spark 工作节点容器分配的内存限额。此设置可以通过 `docker run` 的 `-m` 选项来完成。

```
$ docker run -d -p 8081:8081 -m 256m --name worker-1 runseb/spark-worker
$ docker run -d -p 8082:8081 -m 256m --name worker-2 runseb/spark-worker
$ docker run -d -p 8083:8081 -m 256m --name worker-3 runseb/spark-worker
```

你可能已经注意到，你已暴露主机上 Spark 主节点容器的 8080 端口。这样你就可以访问 Spark 主节点 Web 界面了。只要 Spark 主节点容器在运行，你就可以访问此 Web 界面。当 Spark 工作节点都在线之后，你会在仪表盘中看到这些工作节点，如图 10-6 所示。

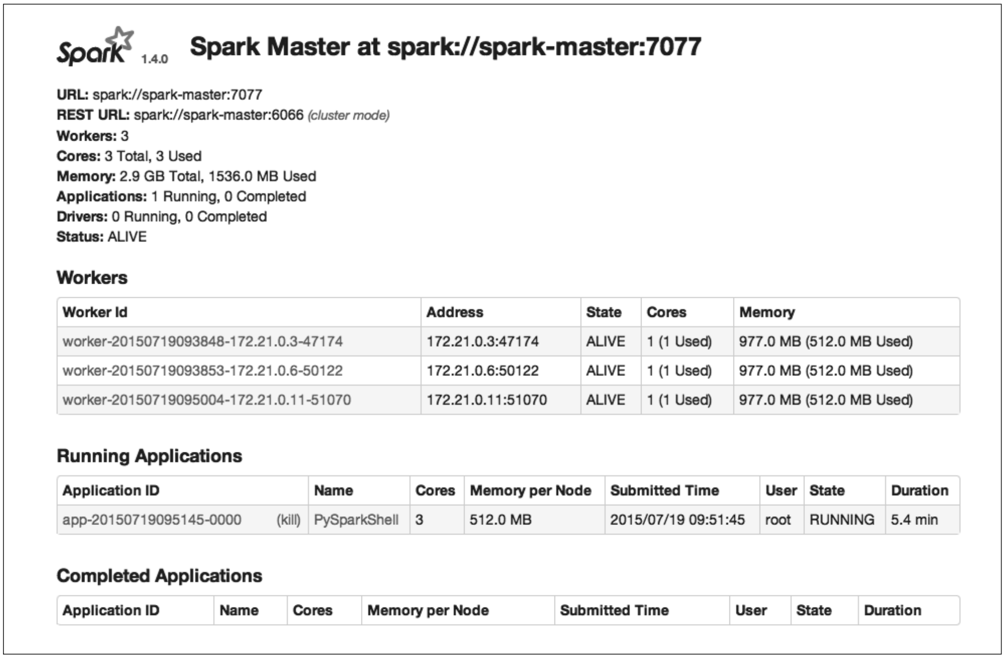


图 10-6：Spark 主节点 UI

这就是全部工作了。这种部署的方便之处在于 Spark 工作节点使用主机名 `spark-master` 来连接主节点。由于 Docker Network 负责管理名称解析，所以每个容器都会自动获得主节点的 IP 地址，并连接到主节点。

10.7.3 讨论

如果你检查一下已经发布的网络服务，你会发现你的四个容器都在 `multihost` 网络上（即 `spark-master`、`worker-1`、`worker-2` 和 `worker-3`）。但是，由于你也发布了主节点的 Web 界面服务，所以每个容器也都被连接到了 bridge 网络上。在下面的例子中，你只能看到工作节点的容器在 bridge 网络上，这是因为 Spark 主节点并没有在运行该命令的主机上运行。如果你在运行 Spark 主节点的主机上执行下面的命令，就会看到 `spark-master` 也连接到了 bridge 网络。

```
$ docker service ls
SERVICE ID          NAME            NETWORK         CONTAINER
92e90b6556b5        worker-1        bridge          ba80b36e5abc
1831b9378d37        worker-2        bridge          c1c8bec01a2a
bc64584793df        worker-3        bridge          f7be3797affb
2bbe00afc559        worker-1        multihost       ba80b36e5abc
7be77369a0ac        worker-2        multihost       c1c8bec01a2a
3a576b7233b6        worker-3        multihost       f7be3797affb
e3c75728c402        spark-master    multihost       fa44cce982df
```

由于公开了 Spark 工作节点的 Web 界面的端口，你也可以通过浏览器来访问这个界面。图 10-7 显示了该工作节点上一个已经完成的任务。

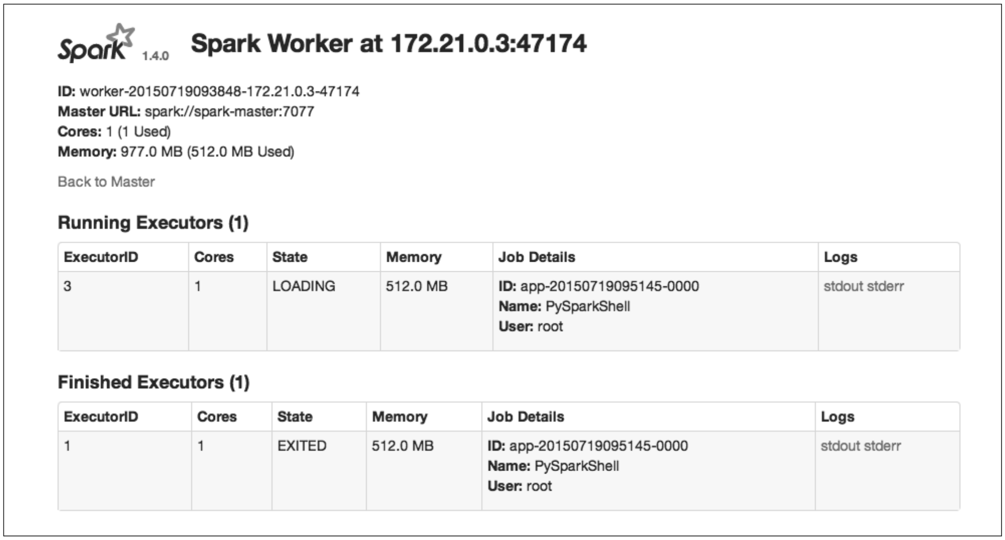


图 10-7：Spark 工作节点 UI

图中仪表盘显示的任务是运行 Spark shell（https://spark.apache.org/docs/latest/quick-start.html）的结果，这是开始学习 Spark 和在容器化的 Spark 集群中运行任务的快速方法。你也可以通过其他交互式容器来运行这个 Spark shell，如下所示。

```
$ docker run -it gcr.io/google_containers/spark-base
root@ac912dd21619:/# . ./setup_client.sh spark-master 7077
root@ac912dd21619:/# pyspark
Python 2.7.9 (default, Mar  1 2015, 12:57:24)
...
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 1.4.0
      /_/
...
>>>
```

由于 `libnetwork` 更新比较频繁，Spark 主节点和工作节点之间的网络连接可能会不太可靠。服务发布机制也可能会发生变化。你应该把这看作是一个进化中的例子，而不是一个可以在生产环境下使用的部署方案。如果遇到什么问题，别忘了使用 `docker logs -f <container_id>` 查看容器的日志。

10.7.4　参考

- 本范例参考了 Kubernetes Spark 的示例（https://github.com/GoogleCloudPlatform/kubernetes/tree/master/examples/spark）

关于作者

Sébastien Goasguen 在 20 世纪 90 年代末攻读博士学位时，构建了他的第一个计算机集群（当时计算机集群还被称为 Beowulf 集群）。从那时起，他就一直致力于计算技术实用化。他研究过网格计算和高性能计算，并在 2005 年前后随着虚拟化技术的出现而转向云计算技术，当时他正在克莱姆森大学做教授。

他目前在 Citrix 公司任职高级开源解决方案架构师，主要从事 Apache CloudStack 项目，帮助开发 CloudStack 生态系统，于 2015 年 3 月当选为 Apache CloudStack 项目的副总裁。他同时也是 Apache libcloud 项目管理委员会和 Apache 软件基金会的成员。Sébastien 专注于云生态系统，并且为数十个开源项目作出了贡献。

关于封面

本书封面上的动物是白鲸，它与独角鲸是一角鲸科的两个种属。

因为习惯在北极生活，所以白鲸在解剖学上不同于大多数其他类型的鲸鱼。它们全身白色，没有背鳍，具有最高百分比的鲸脂；它们的前额凸起，那里是它们的回声定位系统（称为“额隆”）。额隆对白鲸来说非常重要，因为它不仅可以帮助白鲸狩猎，还能够帮助白鲸在移动的冰盖之间找到气孔。

白鲸是高度社会性的生物，它们的群体通常由大约十个成员组成。夏季，这些白鲸群聚集在沿海地区繁殖，这意味着在一个地方可能有数百甚至数千头白鲸。目前世界上的白鲸大约有 15 万头，大多生活在北美、俄罗斯和格陵兰等海域。

北美和俄罗斯的土著人已经有数百年的白鲸捕猎历史了，但在 19 世纪和 20 世纪初期，也出现了商业捕鲸活动。由于捕鲸活动在 20 世纪 70 年代开始受到国际监管，目前只有某些因纽特和阿拉斯加土著部落才被允许继续从事捕鲸活动。

野生白鲸可以存活 70 年到 80 年，在水族馆中它们是很受欢迎的，但其寿命却明显更短。目前，由于栖息地变化、水污染和传染病等原因，白鲸的数量减少，已被认为是一种“频临灭绝”的物种。

O’Reilly 封面上的许多动物都是濒危动物，它们对世界来说意义重大。如果想要了解怎么去帮助它们，可以访问网站 http://animals.oreilly.com/。

封面图片出自 *A History of British Quadrupeds* 一书。

O'REILLY®

Docker经典实例

对于应用程序，无论是在私有云部署还是在公有云部署，本书都列出了丰富实用的解决方案和示例。

即使不具备Docker相关知识，基于书中实用的解决方案，开发人员也能在阅读几章之后打包和部署分布式应用程序。如果你是一位运维人员，你将很快掌握如何使用Docker来改善自己的工作方式。如果你是一位IT专业人士，你将能够学习到如何解决日常问题，比如创建、运行、共享和部署Docker镜像。

- 管理容器、挂载数据卷和容器连接
- 创建和共享镜像
- 单主机或多主机的Docker网络
- 处理Docker配置与开发等进阶问题
- 使用Kubernetes在分布式集群中部署多容器的应用程序
- 使用为Docker优化的新一代操作系统
- 学习用于应用程序部署、持续集成、服务发现和编排的工具
- 在Amazon AWS、Google GCE和Microsoft Azure上使用Docker
- 监控容器，并探讨不同的应用程序用例

Sébastien Goasguen，Citrix高级开源解决方案架构师，主要从事Apache CloudStack项目，帮助开发CloudStack生态系统。他目前是Apache CloudStack项目的副总裁，也是Apache libcloud项目管理委员会的成员。

“这本书对Docker生态系统进行了很好的阐述，非常适合对容器感兴趣的读者。”

——Darren Shepherd
Rancher Labs联合创始人

“这些范例对于帮助我们在自己的平台上快速将Docker运用于生产环境至关重要。”

——Mathieu Buffenoir
Bity CTO

SYSTEM ADMINISTRATION

封面设计：Ellie Volckhausen　张健

图灵社区：iTuring.cn
热线：(010)51095186转600

分类建议　计算机 / 云计算 / Docker

人民邮电出版社网址：www.ptpress.com.cn

O'Reilly Media, Inc.授权人民邮电出版社出版

此简体中文版仅限于中国大陆（不包含中国香港、澳门特别行政区和中国台湾地区）销售发行
This Authorized Edition for sale only in the territory of People's Republic of China (excluding Hong Kong, Macao and Taiwan)

ISBN 978-7-115-44656-5
9 787115 446565

ISBN 978-7-115-44656-5
定价：69.00元

看完了

如果您对本书内容有疑问，可发邮件至 contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：ebook@turingbook.com。

在这可以找到我们：

微博 @图灵教育 : 好书、活动每日播报
微博 @图灵社区 : 电子书和好文章的消息
微博 @图灵新知 : 图灵教育的科普小组
微信 图灵访谈 : ituring_interview，讲述码农精彩人生
微信 图灵教育 : turingbooks